O'REILLY®

TURING 图灵程序设计丛书

高性能Android应用开发

High Performance Android Apps

深入到Android设备的硬件和平台，直击应用层App性能问题，为更好的用户体验提供实战指导



[美] Doug Sillars 著

王若兰 周丹红 夏恩龙 陈文超 李欣欣 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

王若兰

毕业于华中师范大学，现就职于美团网酒旅事业群Android研发组，主要负责美团App和点评App旅游频道的性能调优。

周丹红

毕业于东北大学，现就职于美团网酒旅事业群Android研发组，主要负责大交通频道的开发与性能调优。

夏恩龙

毕业于大连理工大学，现就职于猫眼电影Android研发组，主要负责美团App电影频道的性能调优。

陈文超

毕业于电子科技大学，现就职于猫眼电影Android研发组，研究生期间主要研究qs监控系统App，用于识别人的行为动作，对图片加载、上传和下载的优化有较深的理解和应用。

李欣欣

毕业于哈尔滨工业大学，现就职于猫眼电影Android研发组，爱生活、爱艺术、爱代码。

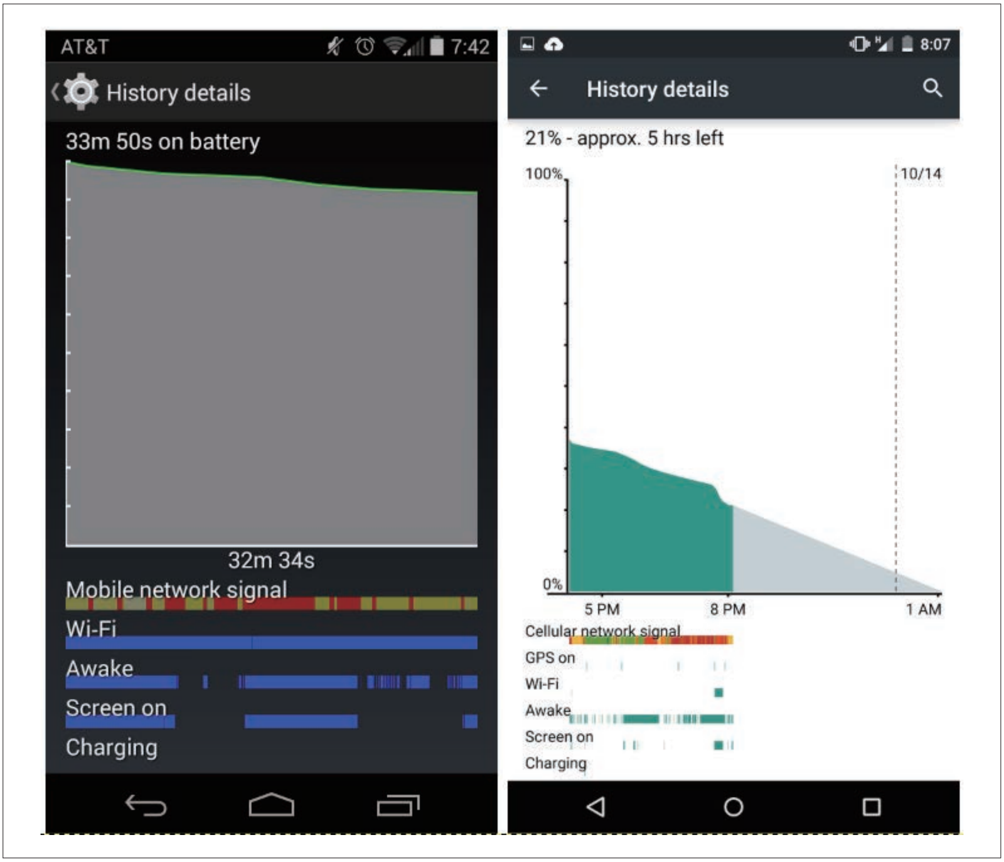


图 3-2：KitKat（左）和 Lollipop（右）的能耗详情

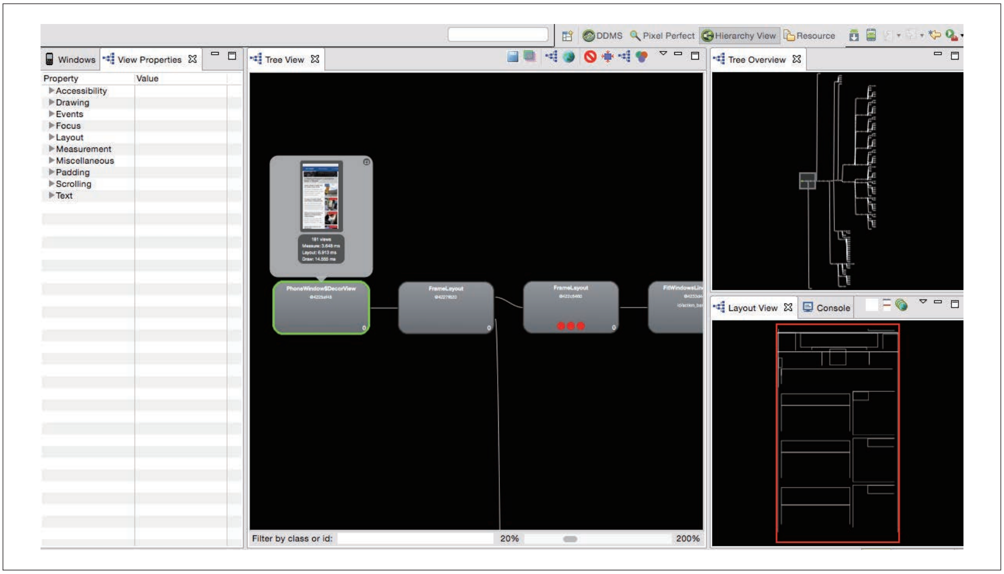


图 4-2：使用树形视图分析一款新闻 App 时，Hierarchy View 工具的界面概览

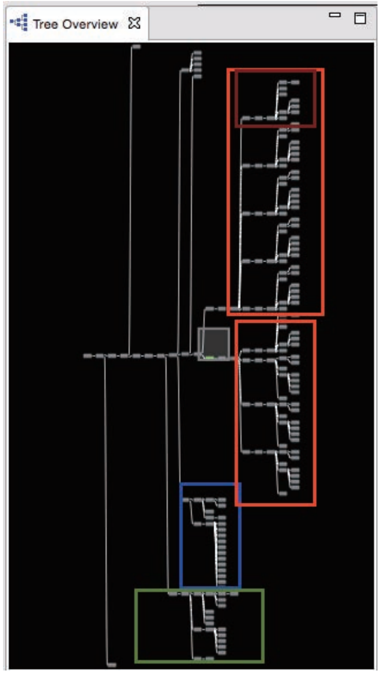


图 4-4：Tree Overview

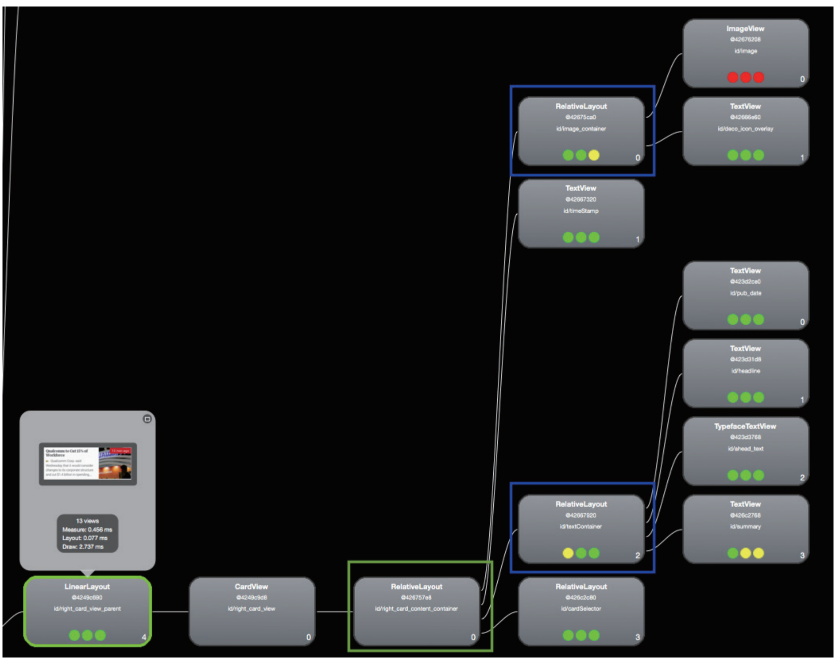


图 4-5：分析一个视图树



图 4-8：未优化的“Is it a goat？”App 的 Hierarchy View

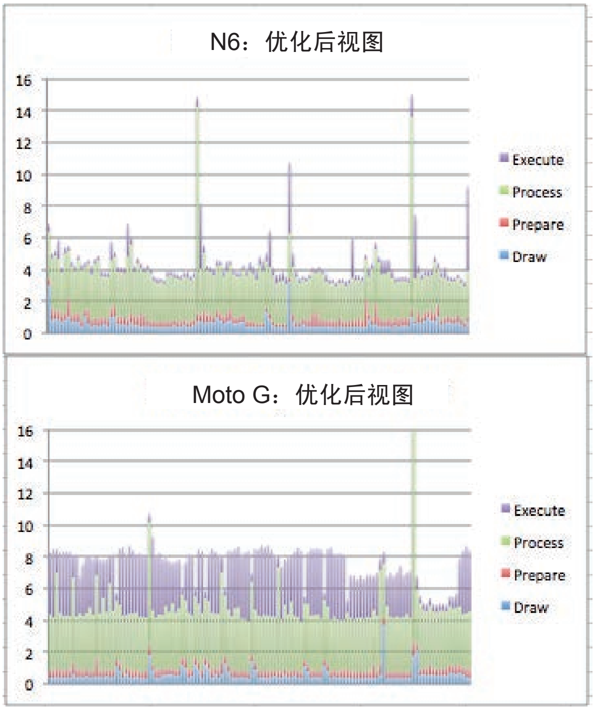


图 4-19：在 Lollipop（上）和 KitKat（下）上优化后视图的 GPU 概况

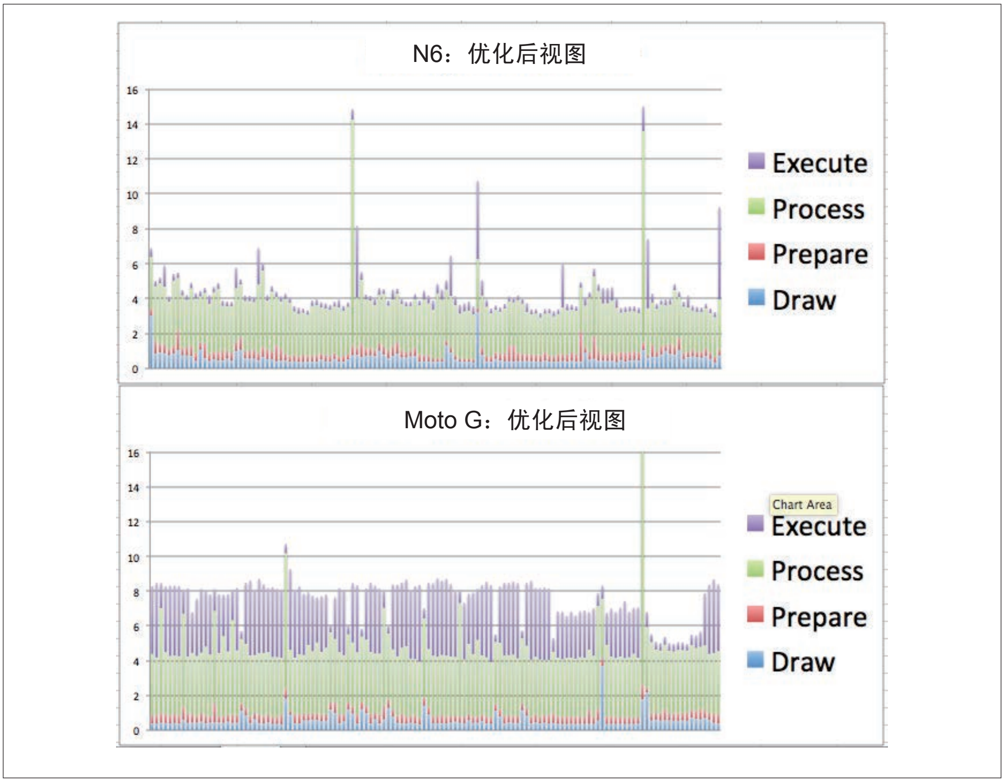


图 4-20：运行 gfxinfo framestats 命令得到的数据

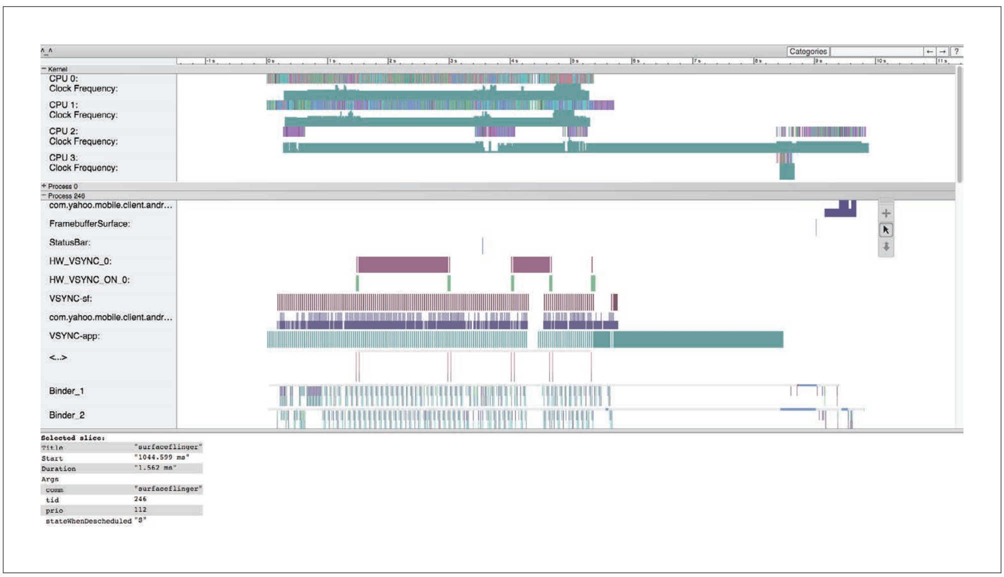


图 4-23：Systrace 的界面（Lollipop）

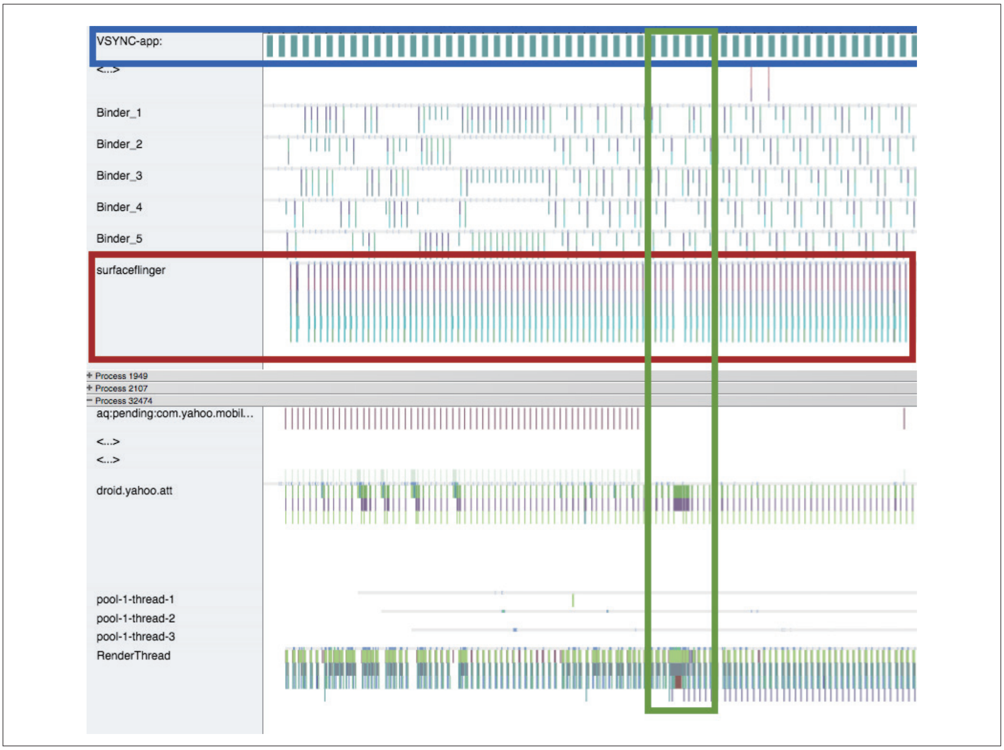


图 4-24：细看 Systrace 的卡顿部分（Lollipop）

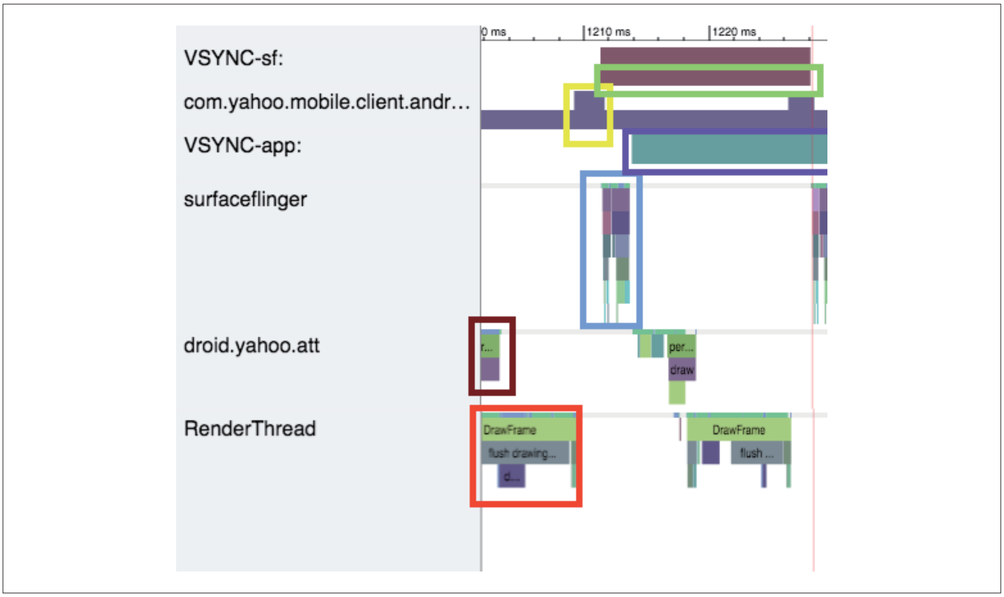


图 4-25：渲染正常的 Systrace（Lollipop）

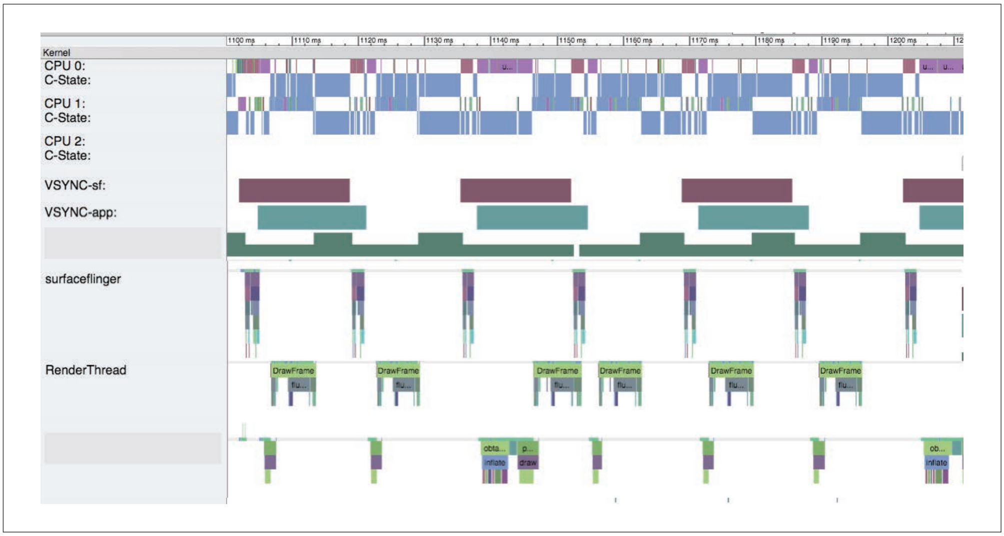


图 4-31：Systrace CPU 信息图

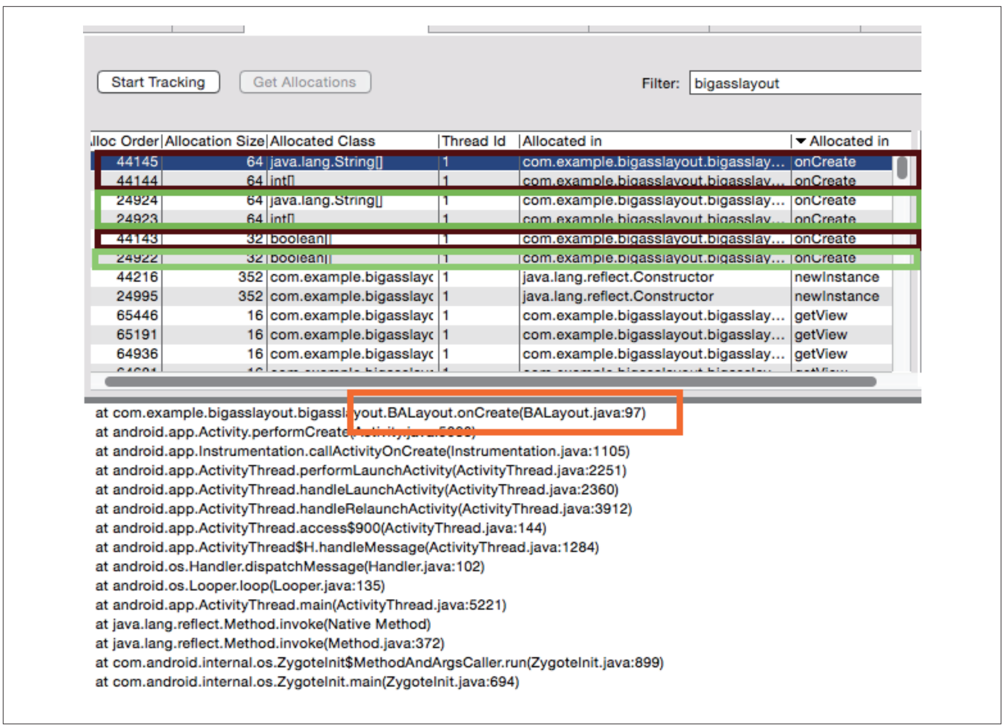


图 5-8：Allocation Tracker 显示的冗余创建的数组

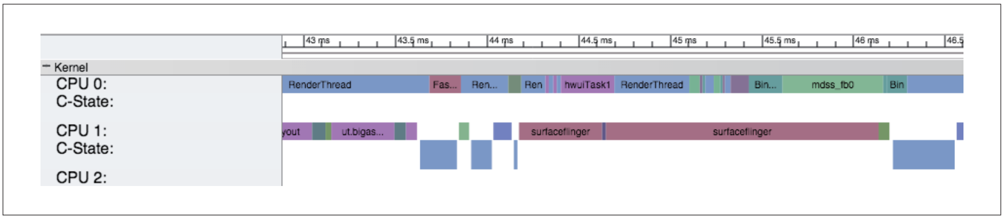


图 6-3：Systrace 的 CPU 视图

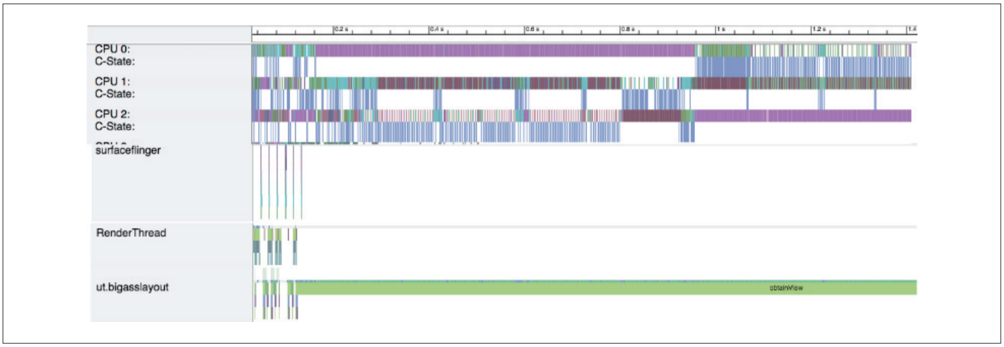


图 6-4：有卡顿的 Systrace 视图

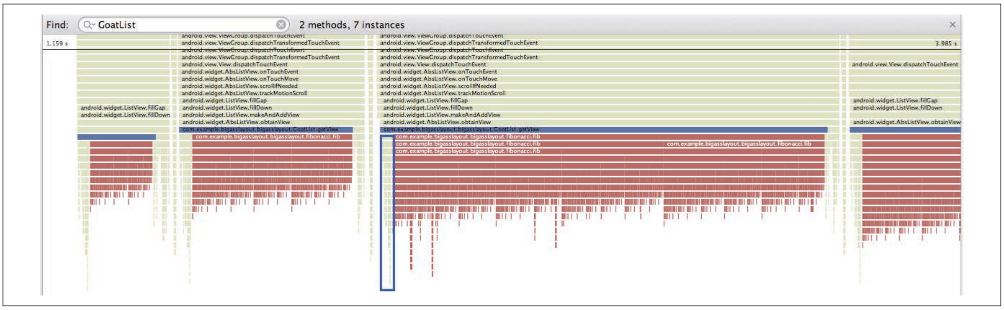


图 6-13：Android Studio 开启斐波那契延迟的 Traceview GoatList 图

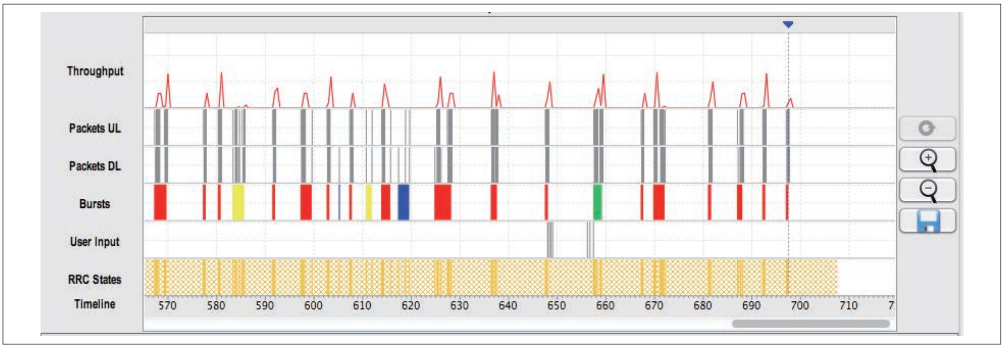


图 7-9：ARO 诊断选项卡数据图

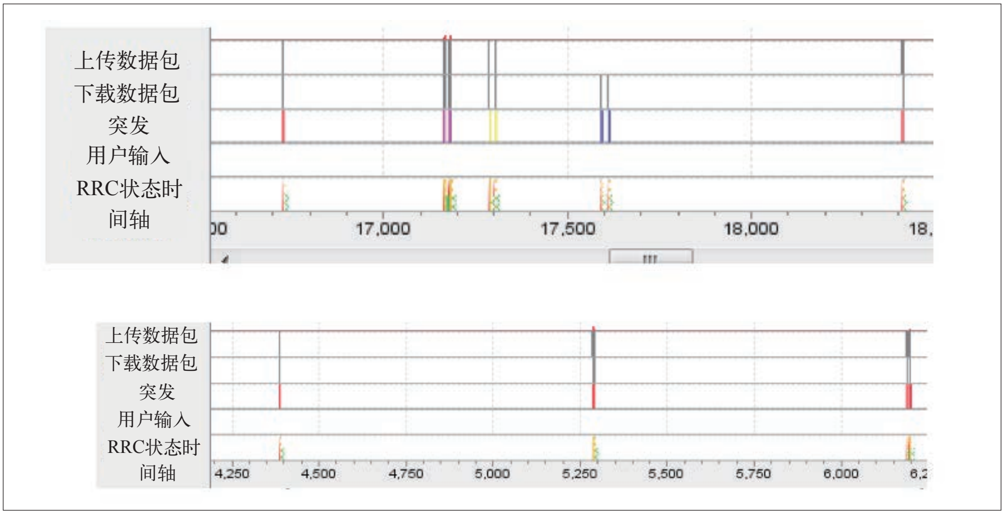


图 7-13：社交媒体后台连接：优化前（上）和优化后（下）



图 8-6：缓慢连接的信息显示板

TURING 图灵程序设计丛书

高性能Android应用开发

High Performance Android Apps

[美] Doug Sillars 著

王若兰　周丹红　夏恩龙　陈文超　李欣欣 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北　京

图书在版编目（C I P）数据

高性能Android应用开发 / (美) 道格·西勒斯
(Doug Sillars) 著 ; 王若兰等译. -- 北京 : 人民邮电
出版社, 2016.10
（图灵程序设计丛书）
ISBN 978-7-115-43570-5

Ⅰ. ①高… Ⅱ. ①道… ②王… Ⅲ. ①移动终端－应
用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2016)第227173号

内 容 提 要

性能问题在很大程度上决定了用户是否会使用一款 App，本书正是 Android 性能方面的关键性指南。全书共 8 章，主要从电池、内存、CPU 和网络方面讲解了 APP 性能优化问题，并介绍了一些有助于确定和定位性能问题所属类型的工具。同时也探讨了 APP 开发人员面临的一些主要问题，进而提出一些可行的补救措施。全书旨在通过提高 App 性能完善 App，以便用户可以获得极致体验。

本书适合与 Android App 开发相关或对 Android App 开发感兴趣的所有人员阅读。

◆ 著　　　　[美] Doug Sillars
译　　　　王若兰　周丹红　夏恩龙　陈文超　李欣欣
责任编辑　朱　巍
执行编辑　杨　婷
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：13.75　　　　彩插：4
字数：319千字　　　　2016年 10 月第 1 版
印数：1－4 000册　　　　2016年 10 月北京第 1 次印刷
著作权合同登记号　图字：01-2016-4630号

定价：59.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序 ······ xi

序 ······ xiii

前言 ······ xv

第 1 章　Android 的性能指标 ······ 1

1.1　性能对用户很重要 ······ 2

1.1.1　电子商务和性能 ······ 2

1.1.2　电子商务之外的影响 ······ 3

1.1.3　性能可以节省基础设备 ······ 4

1.2　最恶劣的性能影响因素：宕机 ······ 4

1.2.1　低性能就像持续的宕机 ······ 5

1.2.2　消费者对性能 bug 的态度 ······ 7

1.2.3　智能手机电池寿命：矿井中的金丝雀 ······ 8

1.3　App 性能问题的检测 ······ 8

1.3.1　模拟测试 ······ 9

1.3.2　真实用户监测 ······ 9

1.4　总结 ······ 9

第 2 章　构建 Android 设备实验室 ······ 10

2.1　你的用户都在使用什么设备 ······ 11

2.2　设备特性分布 ······ 11

2.2.1　屏幕 ······ 11

2.2.2　SDK 版本 ······ 12

2.2.3 CPU/ 内存和存储 12
2.3 用户使用的网络 12
2.4 你的设备不是用户的设备 12
2.5 测试 13
2.6 创建设备实验室 14
2.6.1 你想要花很多钱买设备吗 14
2.6.2 我应该购买什么样的设备 15
2.6.3 除了手机之外 17
2.6.4 Android Open Source Project 设备 17
2.6.5 其他选择 18
2.6.6 其他注意事项 19
2.6.7 我的设备实验室 20
2.7 总结 20
第 3 章 硬件性能和电池寿命 22
3.1 Android 的硬件特点 22
3.2 少即是多 23
3.3 耗电原因 24
3.3.1 Android 能耗统计文件 24
3.3.2 屏幕 26
3.3.3 无线设备 27
3.3.4 CPU 27
3.3.5 其他传感器 28
3.3.6 休眠 29
3.3.7 WakeLock 和 Alarm 29
3.3.8 Doze 模式 30
3.4 基本的电量消耗分析 31
3.4.1 详细的 App 电量消耗分析 34
3.4.2 能耗数据和数据流量 36
3.4.3 App 休眠 38
3.5 高级电池监控 38
3.5.1 电能统计 38
3.5.2 Battery Historian 42
3.5.3 Battery Historian 2.0 51
3.6 JobScheduler 54
3.7 小结 58
第 4 章 屏幕和 UI 性能 59
4.1 UI 性能基准 59
4.2 Android 上的 UI 和渲染性能改进 60

4.3 创建视图 61
4.4 资源缩减 74
4.4.1 屏幕的过度绘制 74
4.4.2 检测过度绘制 74
4.4.3 Hierarchy Viewer 中的过度绘制 77
4.4.4 过度绘制和 KitKat（Overdraw Avoidance） 78
4.5 分析卡顿（测量 GPU 的渲染性能） 79
4.6 丢帧 83
4.6.1 Systrace 84
4.6.2 Systrace Screen Painting 86
4.6.3 Systrace 和 CPU 阻塞渲染 91
4.6.4 Systrace 更新——2015 年 Google I/O 开发者大会 93
4.6.5 第三方工具 95
4.7 感知性能 95
4.7.1 进度条：优缺点 96
4.7.2 动画掩盖加载时间 96
4.7.3 即时更新的善意谎言 96
4.7.4 提高感知性能的建议 97
4.8 小结 97

第 5 章 内存性能 98

5.1 Android 内存：它是如何工作的 98
5.1.1 共享内存与私有内存 98
5.1.2 脏内存与干净内存 99
5.1.3 内存清理（垃圾回收） 99
5.1.4 确定 App 使用的内存大小 102
5.1.5 procstats 107
5.1.6 Android 内存警告 111
5.2 Java 中的内存管理 / 泄露 112
5.3 追踪内存泄露的工具 112
5.3.1 Heap Dump 113
5.3.2 Allocation Tracker 115
5.3.3 增加一处内存泄露 116
5.3.4 更加深层次的堆解析：MAT 和 LeakCanary 119
5.3.5 Eclipse 内存分析工具——MAT 119
5.3.6 LeakCanary 125
5.4 小结 128

第 6 章 CPU 与 CPU 性能 129

6.1 检测 CPU 占用率 130

6.2 使用 Systrace 分析 CPU 131
6.3 Traceview（遗留的监视器 DDMS 工具） 134
6.4 Traceview（Android Studio） 137
6.5 其他优化工具 140
6.6 小结 141

第 7 章 网络性能 142

7.1 Wi-Fi 与蜂窝无线电 142
7.1.1 Wi-Fi 143
7.1.2 蜂窝 143
7.1.3 RRC 状态机 144
7.2 测试工具 147
7.2.1 Wireshark 148
7.2.2 Fiddler 149
7.2.3 MITMProxy 150
7.2.4 AT&T ARO 151
7.2.5 混合型 App 和 WebPageTest.org 154
7.3 Android 网络优化 154
7.3.1 文件优化 155
7.3.2 精简文本文件（Souders：精简 JavaScript） 156
7.3.3 图片 157
7.3.4 文件缓存 159
7.3.5 文件之外 161
7.3.6 分组连接 162
7.3.7 检测应用的无线电使用情况 163
7.3.8 适时关闭连接 164
7.3.9 定期执行重复的 ping 命令 166
7.3.10 网络安全技术的应用（HTTP 和 HTTPS） 167
7.4 全球移动网络覆盖范围 167
7.4.1 CDN 服务器 168
7.4.2 在慢速网络中测试 App 169
7.4.3 仿真慢速网络而不用倾家荡产 169
7.4.4 构建网络感知 App 170
7.4.5 计算延迟 173
7.4.6 最后一英里的延迟 174
7.4.7 其他无线电 174
7.4.8 GPS 174
7.4.9 蓝牙 174
7.5 小结 176

第 8 章　真实用户监测……177
8.1　启用 RUM 工具……178
8.2　RUM 分析：示例程序……178
8.3　崩溃……179
8.3.1　分析 Crashlytics 的崩溃报告……181
8.3.2　使用……186
8.3.3　实时信息……190
8.4　大数据的营救……190
8.5　小结……192
附录　组织性能……193
关于作者……198
封面介绍……198

译者序

相信所有早期的 Android 开发者都被性能问题折腾过。那时这方面的资源几乎搜索不到，更不要提性能优化的最佳实践了。开发者真的是在实战中摸着石头过河，四处碰壁，一身是伤地总结出了很多经验。相信到了现在，每个技术团队都有自己沉淀的一套方法。但时至今日，市面上并没有出现一本指导性的书籍。

当本书的原著刚出版的时候，我们有幸很早就看到了。我们认为这本书非常有价值，有关性能方面的内容非常全面且具有实际的指导意义。当时我们的团队也正在尝试更多的性能优化，于是就决定将这本书翻译出来，让更多的人能够了解和学习到这些经验，并且运用到实际开发中。因此，我们组建了一个翻译小组，一边学习实践，一边翻译这本书。

这里先感谢所有参与到本书翻译过程当中的小伙伴。

酒店部：周丹红、王若兰、杨鑫、罗佳妮、杜航宇。

猫眼部：夏恩龙、陈文超、李欣欣、雷健龙、张涛。

感谢辅助翻译工作的马圣超、高飞、于振兴。

尤其感谢以下五位同学：周丹红、王若兰、夏恩龙、陈文超、李欣欣，他们牺牲了自己的大量业余时间来做这件事情，非常辛苦。

另外，还要感谢后期参与技术指导和校对的同学，找到这些同学的时候，他们毫不犹豫地答应了，为我们提供了大量的建议，同时指正了很多错误。

感谢参与校对的小伙伴（不分先后）：袭建帅、王康、武智、田洪晖、王京。

这里特别感谢刘江老师。我们之前没有太多的翻译经验，是刘江老师为我们提供了很多建设性的建议，让我们少走了很多弯路。

总之，这本书能够顺利出版，离不开大家的努力和帮助。因为能力有限，书中免不了出现一些问题，还请大家包容并给出建议。若对本书内容有任何疑问或建议，可发邮件至：sankuaimj@gmail.com。

序

对于广大的 Android 开发者来说，性能是他们最后才考虑的事情。大多数的 App 开发更强调个性化，开发者的目标是使 UI 看起来完美并且找到一个可行的商业化道路。但是，App 的性能很大程度上像是家里的管道；当它正常工作时，没有人会关注或者考虑到它，然而一旦出错，人们马上就会陷入麻烦当中。

你看，用户在注意到社交小工具、图像过滤器或者是支持克林贡语等其他特性之前，会先注意到 App 的性能不好。并且你猜怎么着？用户因为不满意性能而给 App 差评的比例要高于因其他问题而给 App 差评的比例。

这也是我们说性能很重要的原因。开发 App 的时候，很容易就会忽略性能，但坦率地说，性能涉及你所做的一切。当性能体验不好时，用户就会开始抱怨，进而卸载你的 App，然后报复性地给你一个差评。考虑到这些，性能听起来更像是应该关注的一个特征，而不是必须忍受的一种负担。

但实话实说，提升性能是一件非常困难的事情。仅仅了解算法是不够的，你还需要了解 Android 系统是如何执行它的，以及硬件又是如何响应 Android 系统的操作的。事实上，一行代码有可能会破坏整个 App 的性能，只是因为它滥用了一些硬件限制。但困难不仅仅是这些，因为有时候为了了解后台发生的事情，你甚至必须学习一整套的性能分析工具。这基本上是看待 App 开发的一种全新的方式，并不适合怯于挑战的人。

Doug 写的这本书有什么了不起的地方呢？这本书是 Android 性能方面的实战指南，不仅涵盖了基本的算法话题，还深入到了硬件和平台的工作方式，让你能够了解工具的异常显示是什么含义。这是一本能够帮助工程师转换视角的书。它不再只是关注视图和事件监听器，而是慢慢转换为理解内存边界和线程问题了。

凌晨 4 点，你的 App 运行状况不好，咖啡机也坏了，并且创业孵化器室里有股烂白菜的味道；为了确保上午 10:00 同风险投资者的会议能够顺利进行，你应该看看这本书。祝你好运！

——Colt McAnlis，资深布道师，谷歌公司团队主管，
Google 的 Android 性能模式系列视频的讲师（https://goo.gl/4ZJkY1）

前言

你正在构建一个 Android App 吧（或者你已经构建了一个 App）？你肯定对自己 App 的性能并不满意。（不然你为什么要看这本书呢？）揭示移动 App 的性能问题是一个持续性的工作。我的研究发现，98% 的 App 存在潜在的性能改进空间。本书将涵盖移动性能的隐患，并为你介绍一些测试这些问题的工具。我的目标是帮助你获得这些必要的技能，在重大的性能问题影响到用户之前捕获它。

研究表明，用户期望移动 App 能够快速加载，迅速响应用户的交互，并且在视觉上很流畅、美观。随着 App 变得更加快速，用户的参与度和收益也在增长。没有关注性能的 App 的卸载率和那些会崩溃的 App 的卸载率相同。那些资源利用率低的 App 会造成不必要的电池消耗。运营商和设备制造商收到用户投诉最多的就是电池寿命了。

在过去的几年里，我和成千上万的开发者谈过 Android App 的性能问题。很少有开发者知道有可用的工具能够解决他们遇到的问题。

明确的共识是：运行快速、流畅的移动 App 会更多地被使用，能够为开发者带来更多的收益。令人惊奇的是，哪怕知道这些，很多开发者还是没有使用可用的工具来诊断和定位他们 App 中的性能问题。通过关注性能的提升是如何影响用户体验的，你能够快速地了解你对 App 所做的性能优化工作所带来的收益。

本书读者

本书以 Android 性能为中心涵盖了一系列广泛的主题。任何和移动开发相关的人员都会喜欢本书中关于 App 性能的研究。非 Android 移动开发者将会发现书中关于 App 性能的争论和问题是非常有用的，但用于隔离问题的工具是专门用于 Android 的。

测试人员将会发现用于测试 Android 性能的工具的教程也同样非常实用。

我为什么写这本书

开发者在 Web 性能这个广阔的新兴领域里分享了提高 Web 速度的技巧。Steve Souders 在 2007 年写了《高性能网站建设指南》一书，众多书籍、博客和会议都讨论了这个主题。

此前，移动 App 的性能很少受到关注。App 运行缓慢都被归罪于操作系统或者移动网络，而电量持续时间短则被归罪于设备的硬件。随着手机越来越快，操作系统越来越成熟，用户开始明白 App 对手机性能的影响。

有很多非常棒的工具可以用来衡量 Android App 的性能，但是到目前为止，还没有人对它们进行归纳和整理。通过介绍 Google、Qualcomm、AT&T 以及其他公司的性能测量工具，我希望本书能将 Android 性能测试的奥秘展现出来，帮助你的 App 在不增加用户耗电量的情况下运行得更加快速。

本书预览

当研究 App 性能时，我选择了研究 App 的代码对 Android 设备不同方面的影响。我们将从一个比较高阶的层面开始：性能和 Android 的生态系统，然后查看 App 对屏幕、CPU 以及网络栈等的影响。

- *第 1 章，Android 的性能指标*
  这一章介绍了移动 App 的性能这一话题。我们将用一些例子来证明 App 性能的重要性。文中会强调现在面临的挑战，同样也会列出性能低下在应用市场中的影响。我们还会列出一些统计数据，你可以拿这些数据去说服管理层，让他们在提高 App 性能方面投入更多的精力和时间。这里所给出的数据一般涵盖了所有的移动平台和设备。

- *第 2 章，构建 Android 设备实验室*
  这一章将讨论测试。Android 是一个巨大的生态系统，包括了上万种设备，并且每一种设备都有不同的 UI、屏幕、处理器以及操作系统版本（仅举几例）。我将探索一些方法，帮助你测试尽可能多的设备，并且不会花费过高。

- *第 3 章，硬件性能和电池续航*
  接下来，我们将讨论电池，包括电量流失的原因以及流失的多少。另外，这一章将讨论用户是如何发现 App 中的电量问题的，以及如何使用开发工具来避免这些问题。我们也会学习新的 JobScheduler API（在 Lollipop 版本中发布），它可以从操作系统中唤起 App。

- *第 4 章，屏幕和 UI 性能*
  屏幕是手机上功耗最大的一部分。屏幕是 App 的主要接口，性能差的 App 的卡顿（跳帧）和慢速渲染正是通过屏幕展现出来的。这一章将通过使层级更加扁平化来一步步优化 UI，然后介绍如何使用 Systrace 等工具对 App 进行卡顿和抖动的测试。

- 第 5 章，内存性能；第 6 章，CPU 与 CPU 性能

  我们在这两章讨论内存和 CPU 问题，如垃圾回收、内存泄露，以及它们是如何影响 App 性能的。你将学会如何运用测试工具，如 procstats、内存分析工具（MAT）和 Traceview，剖析 App 以发现潜在的问题。

- 第 7 章，网络性能

  我们将在这一章讨论 App 的网络性能。我们从这里开始探讨移动性能优化，探究 App 是如何与服务器进行通信的，以及我们应该如何加强这些通信。然后介绍如何测试 App 在慢网上的性能，因为许多发展中国家未来几十年用的可能都是 2G 和 3G 网络。

- 第 8 章，真实用户监测

  最后，我们将会讨论如何使用真实用户监测和分析数据，从而保证每台设备都能得到最佳用户体验。如第 2 章所述，无法测试实验室外的每台 Android 设备，但是你可以通过用户设备上的用户监控来获取数据。

- 附录，组织性能

  在此附录中，我们将讨论组织性能，包括如何开始支持构建高性能 App。通过分享探索、成功的案例和概念验证，你可以向公司证明，将性能作为公司追求的目标之一可以改善公司的盈利状况。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/dougsillars/HighPerformance-AndroidApps 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如：“*High Performance Android Apps* by Doug Sillars (O’Reilly). Copyright 2015 AT&T Services, Inc., 978-1-491-91251-5.”

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。

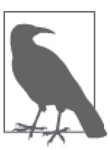
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`Constant width italic`*）
  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：
http://www.oreilly.com/catalog/0636920035053.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

感谢 AT&T 公司的所有人——我的老板、Ed（以及他的上司 Nadine），以及 AT&T 开发项目组的所有人（Ed S.、Jeana 和 Carolyn）。尤其要感谢 ARO 团队的成员：Jen、Bill、

Lucinda、Tiffany Pete、Savitha、John 和 Rod（当然还有所有开发人员），他们每天都会和开发者社区分享自己的性能工具使用心得。感谢 AT&T 实验室以往和现在的所有同事：Feng、Shubho 和 Oliver。感谢你们带来了 ARO 团队的理念，并让我们参与到 App 性能的工作中。

非常感谢早期阅读本书的所有人，你们的评论、提示和建议是无价的。感谢我的技术审阅人和编辑，谢谢你们所有中肯的反馈和建议。你们使得本书更加出色。

最后，同时也是最重要的，我要感谢我的妻子和三个孩子。感谢你们在本书从最初的一个想法到最后成型的过程中，对我工作到深夜（以及次日早上脾气暴躁）的忍耐。没有你们，我不可能完成这项工作。我非常爱你们。1437313。

有意思的是，我是化学反应力学和动力学（研究反应的原理，以及如何使反应加速）博士。没想到，我的事业竟转变为研究移动 App 的机制、优化和动力学！

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

Android的性能指标

说到性能，不同的领域有不同的标准。对于移动 App 领域，性能的评判标准主要体现在 App 的运行方式、工作效率以及使用的流畅性上。在本书中，我们将从提升 App 的效率和速度的角度来探讨性能问题。

众所周知，Android 性能问题非常复杂，因为有成千上万种计算能力参差不齐的设备。但是大多数时候，我们只是保证了开发的 App 能够在自己的目标设备上完美运行。希望本书可以帮助你进一步完善 App，使其能够在 19 000 种不同的 Android 设备上运行并且获得极致的体验。

本书将重点介绍电池管理、工作效率和速度这几个方面的性能优化问题，同时也会探讨开发人员面临的一些主要问题，还会介绍一些能帮助我们确定和定位性能问题所属类型的工具。一旦确定问题，我们将围绕其讨论一些可行的补救措施。

本书适合所有的 Android App 开发者。不论是性能方面的技术带头人、独立开发者、开发团队还是测试人员，都将会从后面章节中讨论的各类性能监测工具和技术中获得帮助。

关于代码优化的建议，每个人所遇到的情况是不同的。有些优化修改简单快速而且效果明显，而有些优化则需要很多额外的工作，比如代码重构、改良 App 的主要架构等。这些优化的建议也许不总是可行的，但是知道 App 到底哪里存在缺陷将能够帮助你持续地提升 App 的性能。

通过研究 App 的基本性能问题，你就可以在感觉到 App 出现问题的时候有个大致的解决方向，并且知道该如何一步步地提升 App 的效率、性能和速度，从而避免 App 的卡顿和用户的投诉。

1.1 性能对用户很重要

你的 App 运行得有多快？早在 20 世纪 60 年代关于人类注意力的相关研究就指出，在 100 毫秒内的行为都被认为是即时的，延迟 1 秒及以上就会让人意识到卡顿 [1]。而延迟和卡顿（即便这种缓慢是感觉上的）对于一个 App 来说是致命的，甚至有时候还会给用户的手机带来灾难（2012 年的一项研究表明，App 出现的卡顿导致 4% 的用户气得摔坏了手机！[2]）。

1.1.1 电子商务和性能

假设一个电子商务类 App 的一个购物流程需要的平均时间长达 5 分钟，并且每个页面的加载平均需要 10 秒钟，那么完成一次购买你最多只能加载 30 个界面到屏幕上。如果能将每个页面的加载时间都减少 1 秒钟，你就可以在每次购物流程中增加 3 个页面。这样可以让用户将更多的商品加入自己的购物车，或者是单纯地将整个交易流程缩短 30 秒钟。

以上的假设其实都是有据可依的。2008 年的一项研究表明，相比那些反应迅速的网站，反应迟缓的网站的交易率和用户满意度更低 [3]。

事实上，前文中虚构的电子商务网站的优化例子和图 1-1 里的数据是基本匹配的。给交易流程平均增加 3.3 个界面，也就等于在整个交易过程中增加了 11% 的页面视图。

注 1：Jakob Nielsen, “Response Times: The 3 Important Limits,” excerpt from Usability Engineering (1993), http://www.nngroup.com/articles/response-times-3-important-limits.

注 2：Mobile Joomla!, “Responsive Design vs Server-Side Solutions,” December 3, 2012, http://www.mobilejoomla.com/blog/172-responsive-design-vs-server-side-solutions-infographic.html.

注 3：Roger Dooley, “Don't Let a Slow Website Kill Your Bottom Line,” Forbes, December 4, 2012, http://www.forbes.com/sites/rogerdooley/2012/12/04/fast-sites/.

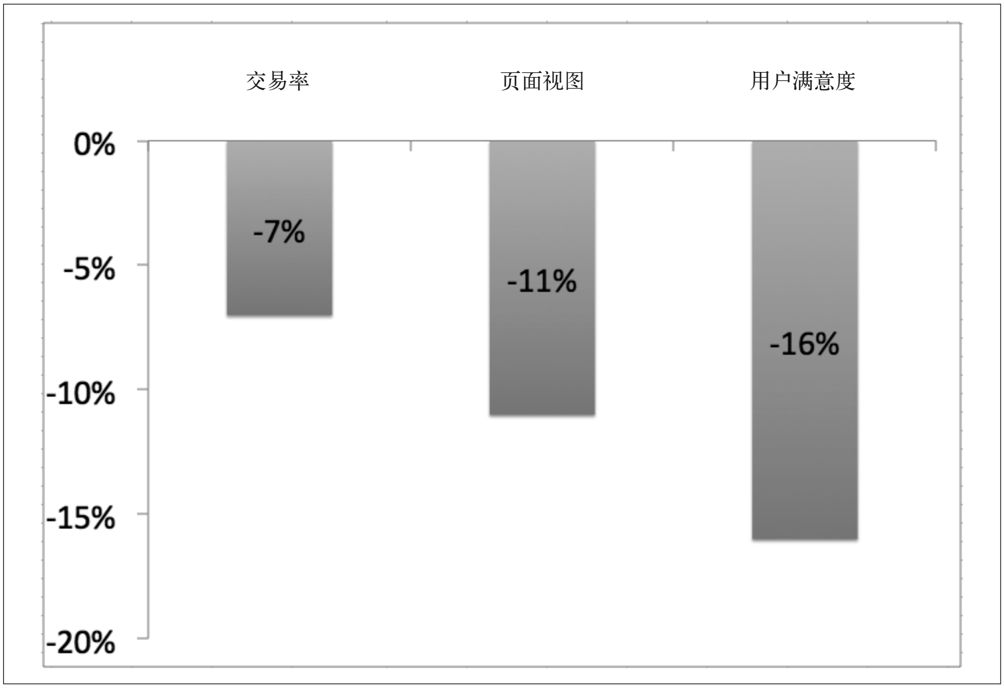


图 1-1：网站速度慢所造成的影响（当网页存在 1 秒延迟时）

对网页性能的研究为移动 App 的性能研究提供了很大的启发。很多研究表明网站反应速度的提升有利于提高订单量和销售量。我认为这同样适用于移动 App（而且考虑到移动端的及时性，这种影响的估计甚至可能还是保守的）。

亚马逊[4] 和沃尔玛[5] 都曾分别报告过类似的数据。这两大零售商都已经意识到，仅仅 100 毫秒的延迟就可能导致他们的收益下降 1%。Shopzilla 为了优化性能重构了他们的网站，结果他们的页面浏览量增加了 25%，转化率也提升了 7%~12%，且重构之后只用了原来一半的节点！[6]

1.1.2　电子商务之外的影响

性能差的 App 不仅会导致销售量和收益降低，还会导致其在 Google Play 市场的排名下滑。更糟糕的是，这些 App 的表现通常会使用户把它们从设备中移除。比如在 2011 年，

注 4：Todd Hoff, “Latency Is Everywhere And It Costs You Sales - How To Crush It,” High Scalability, July 25, 2009, http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

注 5：Joshua Bixby, “4 Awesome Slides Showing How Page Speed Correlates to Business Metrics at Walmart.com,” Radware, February 28, 2012, http://www.webperformancetoday.com/2012/02/28/4-awesome-slides-showinghow- page-speed-correlates-to-business-metrics-at-walmart-com/.

注 6；Todd Hoff, “Latency Is Everywhere.”

T-Mobile 就要求 Google 从应用市场（当时还叫 Android Market）移除一款第三方语音邮件 App——YouMail，因为 YouMail 检查新语音邮件的方式竟然是每隔 1 秒从服务器拉取一次数据（是的，你没听错，一小时就要向服务器请求 3600 次）！如此频繁的连接请求只需要大约 8000 个用户就能产生比 Facebook 还要多的连接数。事实上，在 Google 云推送被广泛地应用之前，这样的做法一直存在。即便是现在，Google Play 上依然存在有类似行为的 App。正如我们所知道的，这些 App 会让服务器、网络的性能变差。最重要的是，它们还会影响用户的 Android 手机设备。

有时候你会发现，刚发布的时候 App 的架构很好，但是后来它变得更大的时候还会如此吗？假设你可以在下届 Super Bowl 比赛中为你的 App 做个广告，App 和服务器架构准备好迎接用户数量呈指数型增长了吗？

1.1.3 性能可以节省基础设备

大多数的 Android App 和远程服务器有高频率的交互，需要从服务器获取很多的内容。减少向服务器发出的请求次数（或是减少每次请求数据的大小）可以让 App 速度得到巨大的提升，而且可以减少服务器端的请求堵塞，节省基础设施上的投入。我曾经所在的公司通过减少 35%~50% 的网络请求和 15%~25% 的数据交互，每年能够在远程服务器设备购置上节约数百万美元。

1.2 最恶劣的性能影响因素：宕机

在 2015 年，一项关于财富 500 强公司的研究表明，网站宕机所造成的损失大约是每小时 50 万 ~100 万美元。为避免收入损失，他们斥资引进了数据中心、云服务器、数据库等作为数据备份。回顾过去的十年，各种各样的研究[7]对宕机所造成的损失持续地做出了评估（而且损失是逐年增加的）。其中有两个研究指出收入损失仅占宕机损失的 35%~38%。如果我们用这些研究所得的数据来做图表，会发现在 2015 年宕机一小时会造成每小时 17.5 万美元的收入损失，而且这项损失是持续升高的（见图 1-2）。

注 7：Yevgeniy Sverdlik, “One Minute of Data Center Downtime Costs US$7,900 on Average,” DatacenterDynamics, December 4, 2013, http://www.datacenterdynamics.com/critical-environment/one-minute-of-data-centerdowntime-costs-us7900-on-average/83956.fullarticle;
Martin Perlin, “Downtime, Outages and Failures - Understanding Their True Costs,” Evolven, September 18, 2012, http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html;
AppDynamics, “DevOps and the Cost of Downtime: Fortune 1000 Best Practice Metrics Quantified,” http://info.appdynamics.com/DC-Report-DevOps-and-the-Cost-of-Downtime.html.

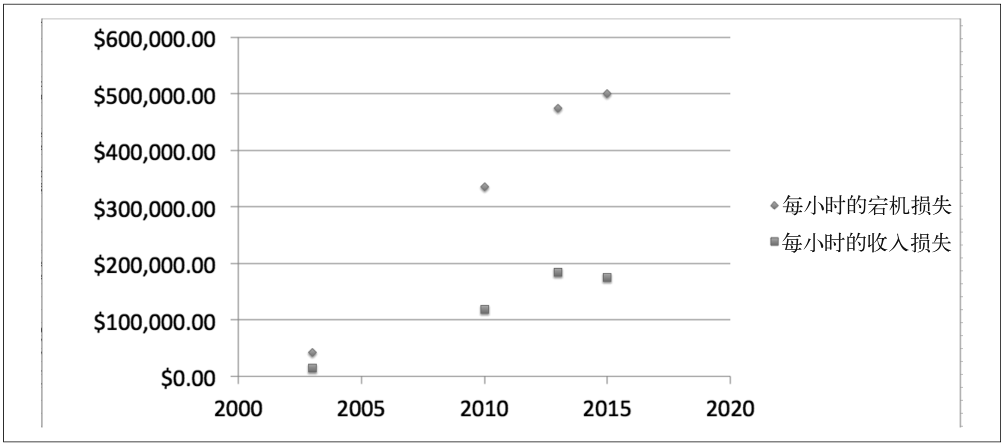


图 1-2：每小时宕机的损失

宕机是最严重的性能问题。一切都将无法运转！也正是由于这样的原因，各家公司每年都会花费数百万美元来保证他们的内容不会被中断。宕机问题除了会损失收入、降低用户满意度外，也会让用户不知所措（见图 1-3）。



图 1-3：洛杉矶郡警察局的 Brink 警佐发的一条推文，回应民众询问警察局 Facebook 宕机的问题

在所有需要改进的性能问题中，保证正常运行对一个公司至关重要。移动端的宕机其实就是 App 崩溃。显然，首要的性能问题就是必须不能崩溃。如果一个 App 都不能正常工作，那它运行得再快又有什么用呢？然而，如果 App 运行得非常缓慢，或者让人感觉缓慢，用户也会有一种遇到宕机的感觉。

1.2.1　低性能就像持续的宕机

电力负载过大时，通常电力公司提供的电压就会比较低，灯光就会显得昏暗（冰箱很可

能无法正常工作)。反应缓慢的 Android App 与此同理。用户仍然可以使用 App，但是滚动可能很卡，图像加载迟缓，整体上的感觉就是慢。就像电压管制对电力公司用户的不利影响一样，一个反应缓慢、性能低下的 Android App 使用起来就像不断反复着宕机状态一样。2015 年 3 月，惠普发布了一份“移动 App 还没能满足用户的预期”报告（http://bit.ly/1OOw5TB）。报告指出，用户对 App 反应迟钝和 App 崩溃一样反感（见图 1-4）。

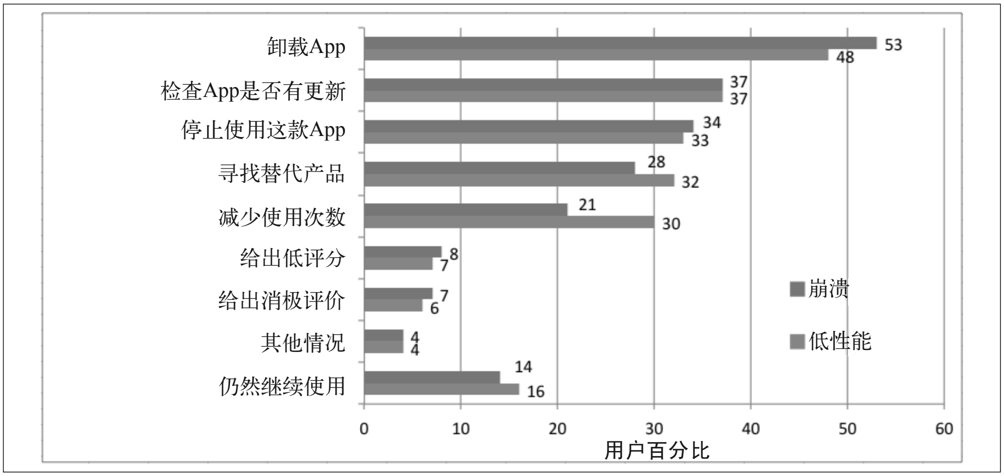


图 1-4：低性能与崩溃：对用户来说是一样的

结合 1.1 节和 1.2 节中的数据，我们可以大致评估出 App 反应迟钝所造成的损失（见图 1-5）。一旦发生宕机，App 就会损失收益[8]。如果知道当加载时间超过 4.4 秒时，转化率就会下降 3.5%~7%，我们就可以大致估算出反应迟钝的低性能 App 至少会每小时减少 6000~12 000 美元的收入。

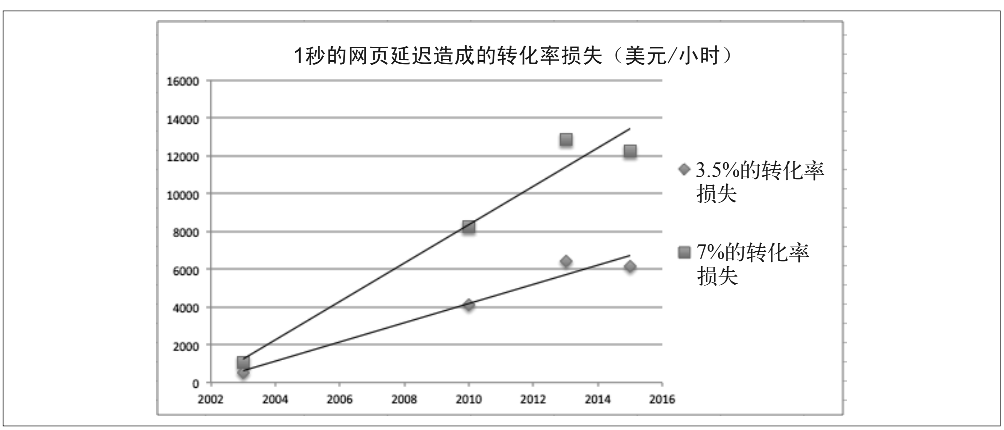


图 1-5：低性能的 App 每小时造成的损失（基于 1 秒的网页延迟）

注 8：当然，有一些用户会在以后再购买，所以实际损失并没有估计的这么多。

如图 1-5 所示，反应迟缓、低性能的 App 造成的损失每年都在增长，App 会逐渐失去收入和用户，最终一无所获——我希望这种悲剧永远都不会发生在你的 App 身上。

1.2.2 消费者对性能bug的态度

由于复杂的开发环境，总会有一些测试过程中遗漏的 bug 影响用户。最近的一项研究表明，44% 的 Android App 的问题和 bug 都是用户发现的，并且这其中的 20% 是用户通过 Google Play 的评价反馈给开发者的[9]。开发者肯定不想通过差评来了解 App 存在的问题。差评不仅反应了某个用户对 App 的低性能的反感，还可能会使看到差评的用户拒绝下载，从而造成用户流失，进而影响 App 的收益。

即使 App 被用户下载了，也不意味着 App 就已经被用户认可了。2014 年，被下载过的 Android App 中有 16% 只被启动了一次[10]。面对选择繁多的 App 市场，用户都是很容易动摇的。如果 App 不能让用户满意，他们很快就会选择另一款同类型的 App。虽然用户卸载 App 的原因很多，但是应该说失望是卸载的首要原因。根据 Perfecto Mobile 的一项研究[11]，让用户感到最失望的问题有以下几个：

- 用户界面问题（58%）
- 性能问题（52%）
- 功能性问题（50%）
- 设备兼容问题（45%）

尽管性能问题在用户卸载 App 的原因榜上排在了第二位，但是可以清楚地看到其他几类问题也和性能息息相关。这足以表明，用户卸载 App 的主要原因仍是 App 的性能差。

如果 App 采用了最小可行产品（MVP）的方法——初始发布时存在 bug 和性能问题——那么当这些问题得到修复后：

- App 依然拥有用户
- 用户会更新 App
- 用户会启动更新后的 App，看看有哪些改进

Twitter 的报告表明，50% 的用户在 3 天之内升级了他们的 App，并且 75% 的用户在 14 天

注 9：Perfecto Mobile, “Why Mobile Apps Fail: Failure to Launch,” Fall 2014, http://info.perfectomobile.com/rs/perfectomobile/images/why-mobile-apps-fail-report.pdf.

注 10：Dave Hoch, “App Retention Improves - Apps Used Only Once Declines to 20%,” Localytics, June 11, 2014, http://info.localytics.com/blog/app-retention-improves.

注 11：查看 “Why Are Your Mobile Apps Failing?(http://info.perfectomobile.com/rs/perfectomobile/images/why-apps-fail-infographic.pdf)” 图表。

之内升级到了最新的版本。[12]Twitter 发现以上情况是反复发生的。所以，只要 App 没有被用户卸载，你的更新版本（bug 修复和性能提升的版本）就有希望：

- 被用户下载
- 被用户打开看到性能的提升

1.2.3 智能手机电池寿命：矿井中的金丝雀

上一小节中的研究表明，用户更喜欢反应迅速、性能优良的 App。智能手机用户最关心的一个问题就是电池的寿命问题。尽管用户还没有普遍意识到，但是 App（特别是没有经过优化的）是耗电的主要原因。我把设置成百分数的剩余电量作为 App 性能的指示器。如果发现电量急剧降低，我就开始审查最近下载的 App 是否存在潜在的问题。

针对新 Android 设备的常见评论是，电池的寿命并没有得到改善。我们认为，如果这些评论者在新旧设备上安装了完全一样的 App，新旧设备的电池寿命就不会有什么明显的差别。迁移到新手机上的 App 依旧在消耗电量。在第 3 章中，我们将展示如何用设备的耗电量来衡量 App 的性能，以及如何提高 App 的性能来为用户节省电量。

移动设备上最耗电的就是屏幕、蜂窝式网络和 Wi-Fi，以及其他的信号发射器（如蓝牙或 GPS）。众所周知，App 必须在屏幕上使用，所以 App 使用移动设备中其他耗电部件的方式才是影响电池寿命的关键。

以往遇到设备的电池寿命问题，消费者通常会指责设备、设备制造商或者运营商。不过现在已经不再这样了。事实上，35% 的消费者曾因为过度耗电而卸载了一款 App。[13] 虽然能够展示 App 耗电量的消费者工具才刚刚进入市场，但是它们的质量在快速地提升。值得庆幸的是，节省电量的开发者工具也已经开始出现了。我们将在第 3 章中讨论这些工具。在设计和开发 App 的过程中，开发者最好认真地对待电量和电源问题。

1.3 App性能问题的检测

在发布 App 之前，发现性能问题的最好方法就是不断地测试。在第 2 章中，为了使 Android 系统有更好的兼容性，我们将介绍应该测试的所有机型。后续的章节将介绍很多可帮你诊断性能问题的工具，以及解决这些问题的技巧。一旦 App 发布到了市场上，就要确保 App 能将用户问题和使用情况反馈给开发者。阅读这些反馈报告并分析其中的信息更有利于开发者解决已发现的问题。

注 12：查看“Scaling Android Development at Twitter”(https://youtu.be/T5qEnillTHc?t=6m43s)，Twitter 的 Jan Chong 在 2014 年巴黎 Droidcon 上的演讲。

注 13：Hewlett-Packard,“Failing to Meet Mobile App User Expectations: A Mobile User Survey - Dimensional Research,”http://bit.ly/1LXYPec.

模拟和真实用户监测（real user monitoring，RUM）是两种常见的性能测试方法。

1.3.1　模拟测试

实验室的模拟测试需要创建特定的测试用例，或者是在 App 上模拟用户行为。后面将要讨论的很多和 App 一起运行的工具在寻找异常的时候都需要测试用例来模拟运行。这是一种发现、解决 bug 和性能问题的好方法。然而，Akamai 给出的每天 19 000 份独特的 Android 用户代理报告指出，[14] 并不是所有的场景都可以用测试用例来模拟。

1.3.2　真实用户监测

并不是所有场景下的测试都可以在实验室进行，所以收集真实的用户性能数据是十分必要的。在 App 中插入分析代码库来收集用户的真实数据，能够让你快速地了解用户可能面对的问题的类型。这样你就可以解决所发现的用户问题和 bug。解决问题之后，想办法将用户问题复制到实验室，这是避免以后的发行版 App 出现此类问题的聪明做法。第 8 章将探讨一些从 RUM 工具中获取到的结果。

1.4　总结

本章中给出的证据强有力地表明，性能好的 App 的加载、滚动和其他用户事件都足够快速平滑。性能差的 App 导致的用户流失率和 App 崩溃差不多。因此性能差和持续宕机都会失去订单量、销售额和用户（不管是现在还是未来）。相信这些证据足以说服你（也足以说服你的经理和领导）。让我们来解决所有的性能问题并让 App 更快地运行吧！

注 14：Alec Heller, “UA Strings Are Terrible: Adventures in Server-side Device Characterization for Site Performance,” Velocity 2014: Building a Fast and Resilient Business, June 25, 2014, http://velocityconf.com/velocity2014/public/schedule/detail/35211.

第 2 章

构建Android设备实验室

Android 生态系统是世界上最大的移动平台（根据市场占有率）。Google 称，世界范围内活跃的 Android 设备数已经惊人地超过了 10 亿。Android 设备占有了约 80% 的智能手机市场份额。在如此大的市场份额下，Android 应用的开发无疑成为了一大热门。然而，Android 生态系统的迅速成长也带来了一些有趣的挑战。

它经历了 12 次重大的版本升级，拥有数以千计的手机型号（还有平板电脑、手表、电视等）和数十种屏幕尺寸，并且制造商对 Android Open Source Project（AOSP）[1] 进行了调整。由于 Android 生态系统的复杂差异性，App 测试不可能覆盖所有的设备 / 系统组合。Akamai 称，他们每天能记录 19 000 个不同的用户终端。[2] 那么，如何才能保证你的应用在有代表性的 Android 设备上运行良好？你又如何知道什么才是*有代表性的设备*？

TestDroid 的一项研究表明，你需要 12 种机型就可以测试全球排名前 20% 的设备。如果你想测试 50% 的设备，那么你需要 60 种以上的机型。例如，在美国市场上，25 种设备占据了约 66% 的市场份额，但是如果你想覆盖到 90% 的设备，那么你需要 128 种机型。因为测试时间通常是很宝贵的，所以你不太可能（除非使用自动化工具）规律性地在如此多的设备上测试。在这一章中，我们会提供一些构建 Android 设备实验室的不同选择，帮助你使用最少的设备，最大化地在功能、性能和界面方面测试你的应用。

注 1：Android Open Source Project 是 Google 发起的 Android 开放源代码项目。——译者注

注 2：Alec Heller, "UA Strings Are Terrible: Adventures in Server-side Device Characterization for Site Performance," Velocity 2014: Building a Fast and Resilient Business, June 25, 2014, http://velocityconf.com/velocity2014/public/schedule/detail/35211.

2.1 你的用户都在使用什么设备

最简单的方法就是查看 Android 不同系统的使用率，因为性能优化在不同的系统上表现各异。例如，2015 年 6 月，只有 12.4% 的 Android 用户在使用 Lollipop 版本，39.2% 的用户使用 KitKat（KK），37.4% 的用户使用 Jelly Bean（JB）。同时，5.1% 的用户仍然使用 Ice Cream Sandwich（ICS），Gingerbread 和 Froyo 累计起来也有 5.9% 的使用量。当新用户涌向最新的设备和最新的系统版本时，仍然有大批的用户停留在三年之前开发的系统和设备上。设备在世界上不同区域的分布也各不相同，高端设备在发达国家销售火爆，二手设备（和低端的新设备）却在发展中国家占据主导地位。

2.2 设备特性分布

Android Gingerbread 至少需要 128MB 的内存和 2GB 的存储空间。我们把它作为设备的下限，因为现在仍有许多这样配置的设备在运行。有些设备没有摄像头，有些只有后置摄像头，有些前后摄像头都有。还有一些设备附带了 NFC、温度传感器、加速度传感器和气压计。可以想象，这些设备有相当大的差别。接下来我们研究一下对开发影响最大的几个因素。

2.2.1 屏幕

屏幕尺寸一直是 Android 开发者最关注的地方，因为如果应用外观差，或者显示不正常，就会被用户嫌弃。像前文提到的，相差悬殊的屏幕尺寸并没有让这种情况好转。确保 App 在所有尺寸的屏幕上都能正确显示，是开发过程中决定性的一步。近几年来，美国市场上设备的屏幕尺寸越来越大，他们并不在意手是否拿得住或裤兜是否装得下。Samsung Galaxy S（2010）高 122mm，屏幕分辨率是 480×800。而 S5 的高度已经达到了 145mm，分辨率为 1080×1920（在短短四年内高度几乎增大了 1 英寸，像素密度达到了原来的 5.4 倍）。最新的 Motorola Nexus 6 将巨屏手机的屏幕高度上限提高到了 151mm（5.92 英寸），Quad HD 分辨率达到了 2560×1440 。

虽然屏幕越来越大，但仍有一小部分用户在使用 480×320 分辨率或更低分辨率的设备（根据 MarketingProfs 的一项调查，[3] 这一比例在 2014 年第二季度超过了 5%），17% 的南非用户正在使用 240×320 分辨率的电话。在测试实验室中，开发者很希望只使用超大的、绚丽的屏幕，但是保留一两款小屏设备仍然是一个明智的选择。如果预算不允许，可以用模拟器代替小屏手机来测试 UI 的工作情况。但需要注意的是，模拟器并不能准确地反映真实设备的性能。

注 3：MarketingProfs,“Mobile Trends: Most Popular Phones, Screen Sizes, and Resolutions,”http://www.marketingprofs.com/charts/2014/25740/mobile-trends-most-popular-phones-screen-sizes-and-resolutions#ixzz3hPrTffs4.

2.2.2 SDK版本

不同 SDK 版本的设备可能在组件和性能上有相当大的差异。Jelly Bean 版本之后的手机受益于“Project Butter”，该项目使手机在 UI 滑动和渲染上更加流畅，卡顿更少。KitKat 版本包含了“Project Svelte”，减少了操作系统的内存使用，以至于在 512MB 内存的设备上也可以运行（甚至只有 256MB 内存的、运行 KitKat 的设备也曾经进入过 Google Play 市场）。Lollipop 引入了“Project Volta”，通过更新 SDK 减少电量消耗。虽然这些升级令人非常激动，但同时也使得这些版本之前的设备开发复杂化。

虽然很多设备在被使用了较短时间（6~18 个月）之后就被丢弃，但仍然有很多 Android 手机的使用时间长达两年以上。在 2015 年的市场上，仍然有全新的 Samsung S3（2012 年发布）在售卖（与它之后的版本 S4、S5、S6 一起），并且位居使用榜上前五名，直到现在仍然在国内外的市场上火爆销售。

2.2.3 CPU/内存和存储

2014 年 10 月，印度排在前两名的手机都运行 Jelly Bean 系统，配备了单核 CPU、512MB 内存和 4GB 的内置存储。在中国，高端设备与美国和欧洲流行的机型差不多，比如运行 Gingerbread 的单核设备。App 可能在美国的普通设备上运行良好，但是它如何应对性能更低、内存或存储空间更少的设备呢？当 App 达到了设备允许的最大内存时，它会停止分配内存，还是继续运行（拖慢性能并且容易引起崩溃）？

2.3 用户使用的网络

我们会在第 7 章详细地讲解这个问题，但是在北美（尤其在城市里），我们已经用上了高速的 LTE（最新的研究表明，97% 的人已经用上了 LTE 或其他的 4G 技术）。在西欧，这一比例降至约 83%。在世界的其他地区，这一比例会更低。很多地区甚至还没有 3G，仍然在使用老旧的 2G 网络。如果你计划让你的 App 在世界上大部分地区上市，你应该在不同网络条件下测试 Android App。第 7 章会提到模拟慢速网络的方法，这样你就可以保证无论用户在哪、使用何种网络，你的 App 都将运行顺畅。

2.4 你的设备不是用户的设备

“只用我自己的手机测试”的时代已经结束了。当我参加开发者大会的时候，所有与会者都有一部高端手机——通常是最近一两年的旗舰款。拿 2015 年来说，开发者都拿着和 Nexus 6、Sumsung S6 或者 HTC One（M9）相同配置的设备。这些设备全都运行 Lollipop 系统，拥有 4 个或 8 个核心的 CPU、几百万像素的摄像头、16~32GB 的存储空间、2~3GB 的内存，并且可以录制高清视频。这些设备非常棒，而且很有趣。但需要注意的是，

Android 开发者大多生活在科技中心，使用的是快速的 Wi-Fi 和蜂窝网络，意识到他们使用的设备并不是 Android 阵营里的主流机型是尤其重要的。

10 亿活跃 Android 设备运行在迥异的网络环境下，但开发者都生活在高端设备和高速网络的摇篮里。移动数据在发达国家持续增长，但是已经开始趋近饱和。新用户增长最快的地区将是发展中国家。接下来的 10 亿连接互联网的用户关注的是便宜的设备。满足这些用户需求的 Android 设备将具有巨大的市场潜力。这些设备和你的设备有明显的区别（它们通常和你几年前淘汰的设备或你借给家人然后忘记的设备差不多）。这些设备缺少我们习以为常的性能，但这不是我们在满足这些用户时要考虑的唯一因素。我们还必须了解他们所面对的局限，如手机充电时的通电问题和移动网络供应数据的质量问题。

> **root 设备 / 工程版本 / 开发者版本**
>
> 被 root 的设备是指那些获取了 Android 内核 root 权限的设备。很多开发者都是爱好折腾的玩家，他们喜欢获取 Android 内核的 root 权限。作为一名开发者，你应该知道你的 App 可能会在被 root 的设备上运行，并且你应该准备好应对被 root 的设备上可能产生的安全问题（例如用户查看任何文件，甚至是被保护的沙箱中的敏感内容）。
>
> 被 root 的 ROM 通常安装了一个超级用户 Android 应用包（APK）以作为 App 和内核之间的接口（如果你没有的话，Google Play 中有很多不错的选择）。
>
> 开发者版本 / 工程版本是被 root 设备中的一部分。root 社区把这些版本称为“不安全的”版本。这是因为调试被开启，安全选项就会被关闭。在工程版本上，你可以查看并且运行任何 App。这是一个非常强大的选项，并且对 Android 开发者很有用处。另一方面，许多 Android 应用有严重的安全漏洞问题，在 root 权限下它们会变得更加严重。对于测试而言，使用 root 权限测试给了你接近 Android 系统内核的更大权限。出于同样的原因，个人使用时，你应该谨慎使用一个被 root 的设备。
>
> 对于本书中后续讨论的一些测试工具来说，root 权限提供了一些额外的、有用的特性。它可能对安全性测试也有帮助（超出了本书的范畴），因此，我建议保留一部具有 root 权限的设备以便测试。
>
> 免责声明：在一些地区，获取设备的 root 权限会导致法律或版权问题。在美国，只要没有侵权问题，获取 Android 手机 root 权限目前是合法的（但是平板电脑不行）。如果你不清楚这件事在当前地区的合法性，请咨询你的法律顾问。

2.5 测试

考虑到上述困难，如何为 10 亿用户、大约 2 万台设备的 Android 生态环境创建一个可控制的测试环境呢？如何保证我们在支持发达国家用户的同时，也为全世界范围内的潜在用户提供足够的支持？现在，我们很清楚已经不能用“口袋里的设备”来解决这个问题了。

在 2.6 节中，我们将讨论一些方法，这些方法可以覆盖现有设备并且会为以后提供支持。显然，每个人的预算相差甚远，所以，在考虑这些建议时，根据你的能力添加（不要减少太多预算）预算吧。

2.6 创建设备实验室

确保 App 做了符合预期的事的唯一方法就是测试，并且尽可能覆盖多种尺寸的屏幕和配置。为此，你需要一个测试用的 Android 设备实验室。

你的设备实验室可能只是一个办公桌抽屉，里面装了各种处于充电状态的设备，它们在一堆电缆中纠缠着。这样做的好处是设备都触手可得，并且能够保证设备安全。但是，缺点更明显：只有你可以访问你的"实验室"，你所需要的设备可能没有充电，最重要的是"眼不见，心不想"。如果你不经常看你的设备，可能会忘记测试它们。

还有一种方法就是开放设备实验室。这样不仅能够保证设备安全，还能方便人们使用、注销和测试。

当需要测试设备时，确保覆盖绝大多数用户并保持预算是至关重要的。如果你已经有一款 App 在应用市场上架，就很容易通过数据分析出用户使用设备的分布（关于 App 分析的详细内容，见第 8 章）。也许用户设备的分布和最主流设备的分布有所偏差。为了保证现有的用户高兴，你应该确保一直使用用户最多的设备测试。如果你没对 App 作分析，就不得不看热门 Android 设备的报告了（通常这些报告都大同小异，所以你用这些数据来选择设备也是比较保险的）。

2.6.1 你想要花很多钱买设备吗

开销一直是个大问题。在这一小节中，我们将讨论一个理想的 Android 设备列表是怎样的，但最后还是财务说了算。你可能会被问到："为什么不用模拟器来测试不同的设备？"模拟器可以在很多方面起作用（许多屏幕大小不同的模拟器或许能帮助解决 UI 问题），但是，作为开发者，我们都知道模拟器存在问题（速度问题，不能使用传感器，比如位置和加速度传感器等）。你需要说服你的上司，真机对于性能测试是必需的。或许，在会议上用一个或者三个模拟器进行性能测试，能够帮助你说服上司（Twitter 的 Android 团队就这样做过）。

除了预算之外，你需要用不同的设备来覆盖下面这些参数：

- 屏幕尺寸
  - 小（4.4%）
  - 中（82.9%）
  - "巨屏手机"（8.6%）

- 平板电脑（4.1%）
- 特殊情况（穿戴设备、电视、汽车等）

- 屏幕分辨率
  - 低（4.8%）
  - 中（16.1%）
  - 高（40.2%）
  - 超高（36.6%）
- 处理器
  - 双核
  - 四核
  - 多核
- 内存
  - RAM
  - 存储（例如，对比容量可用很少和大量空闲的设备）
- 网络速度
  - 3G
  - LTE
  - Wi-Fi
- SDK 版本
  - Gingerbread（2.3、2.3.3、2.3.7）
  - Ice Cream Sandwich
  - Jelly Bean（4.1、4.3）
  - KitKat
  - Lollipop
  - Marshmallow（或更高）
- 其他考虑
  - root 设备
  - 安全测试
  - 原始设备制造商（OEM）差别

好消息是我们可以用相对较少的设备匹配这些不同的特性。有很多方法可以把这些特性划分到不同的电话组里。

2.6.2 我应该购买什么样的设备

假设你没有足够的财力（或时间）来测试 100 种手机，让我们找一个方法使用尽可能少的设备来进行更多的测试。有很多购买设备的方式，而且没有比下面提供的几种方式更好的了。

- Facebook 在选择测试设备上走了一条独特的路线。[4] 他们没有选择 Craigslist 和 eBay 网上的旧手机，而是选择了与从 2008 年起历年高端的、特性相近的手机。这使得 Facebook 团队不仅模拟了现在高端手机的用户体验，同时也照顾到了过去几年的热门机型（也可以代表现在在售的低端手机）。2014 年的 Facebook 报告表明，最常见的型号符合“2011 年”的型号，双核，1GB RAM（Facebook 用 Samsung S2 测试这类设备）。他们已经发布了检测设备“年份类型”的开源库（http://github.com/facebook/device-year-class），所以你可以投放适当的内容给使用特定设备的用户。
- Etsy 使用设备分析来推断现在流行的设备，并从这个列表中采购设备。Esty 的设备没有配置全系的电池，所以可以在老设备上发现更真实的电量消耗。当新设备发布时，Etsy 团队会观察哪些设备在用户中快速增长，然后相应地调整他们的测试设备。

其他提示：如果 App 需要做很多的计算，你要用不同的 CPU 类型测试。如果有很多渲染，那么你要关注那些大屏的但是配置了低性能 GPU 的设备，因为这可能成为性能瓶颈。在后面的章节中，我会讨论 SDK 的改变如何从不同方面提升性能，所以，你应该关注那些使用早期 SDK 版本、没有优化过性能的设备。

1. 之前流行的

24、36 或 48 个月之前的高端设备可以很好地匹配“老设备”的标准。你可以从 eBay 或 Graigslist 淘到这些设备，这些网站可以帮助你用很少的钱获得这些小屏幕的老版本 SDK 设备。

Nexus S 可以运行从 Gingerbread 到 Jelly Bean 版本的系统（但是它有一个比较大的屏幕，480 × 800 分辨率），所以，为了扩大设备组合范围，最好用像 Samsung Galaxy Y（240 × 320 分辨率）这样低分辨率、小屏幕、使用 Gingerbread 系统的设备。这种方法的测试会存在一些误差，因为测试结果只在你购置的机型上是准确的。

2. 目前流行的

这和你的分析结果相比可能略有不同，但是截至 2014 年，网上多个来源表明，Samsung S3（最早在 2012 年搭载 ICS 系统发布）仍旧是使用最广泛的 Android 设备。另外，Samsung S2（在 2011 年第一季度发布）仍然排在前 10 位。这有力地证明了流行设备持久的生命力。S3 有很多系统版本，比如 ICS、JB 和 KK（S2 只能升级到 JB）。尽管 S3 设备分布趋于稳定（事实上，可能还有略微减少），这款设备仍然会是很长一段时间内的主流设备。添加当前热门的旗舰机型也是很重要的，因为它们会为接下来几年的测试提供帮助。

3. 以后流行的

Nexus 设备（由 Google 直接出售，保留纯粹的 Google 体验，没有原始设备制造商的修改）

注 4：Facebook, Year class: A classification system for Android（https://code.facebook.com/posts/307478339448736/year-class-a-classification-system-for-android/）.

通常不享受运营商补贴，因此不是销售最火爆的设备。这可能导致你不把这些设备当作备选，但它们通常会率先更新操作系统。这样一来，你就可以早于主流设备提前在最新的操作系统上测试 App。比如 Android Marshmallow，Google 最先在 Nexus 5、6、9 上向开发者推送最新的操作系统。因此，为了让 App 将来能适配最新的系统，最好保留一款最近的 Google Nexus 设备。

2.6.3 除了手机之外

除了手机和平板电脑之外，Android 正迁移到其他的生态系统，比如穿戴设备、电视和汽车。这些平台与传统的 Android 设备不同，你可以根据你的开发计划在日常测试中涵盖其中的一部分。

Android Wear

Google 在 2014 年的开发者大会上宣布 Android Wear 是 Android 的一种新形式。运行 Android Wear 的设备通常是 Android 智能手表，其通过蓝牙与 Android 设备（没有电话号码或 SIM 卡）进行通信。Google 建议，App 在手表和手机上应该配有不同的界面。用户接收到的信息应该体现在一系列的卡片上。Google 将交互划分为下面两种：

- *建议*

  及时信息列表（例如，消息、位置相关的数据等）

- *要求*

  允许语音命令控制 Wear 设备发出请求数据

这种开发模式与传统的 Android App 明显不同，所以如果你计划为 Android Wear 构建 App，应该在实验室配备一两个代表性的设备。

2.6.4 Android Open Source Project设备

当我们在讨论 Android 是开源软件的时候，经常漏掉一件事情，即我们习惯说的 Android 的 Google 版本仅仅是 AOSP 的一个分支。在前面对 Android 设备的描述中，我把主要的目标都集中在各种各样的 Google 设备上，因为它们在美国是主要的设备。但是，为了覆盖所有的 Android，我们可以再讨论一下其他常见的 AOSP 分支。

2014 年夏天，研究估计 AOSP 设备占有 20% 的智能手机市场（Google 分支占有 65%）。这些设备缺少以下组件：

- 分发 App 的 Google Play Store
- Google Cloud Messenger 推送消息
- Google Play 服务

- Google 产品，几乎所有 Google 定制的其他工具类 App

然而，这个生态系统并不是可以忽略的，所以你应该把这些设备也作为 App 分发策略的一部分。

1. Amazon

在美国最常见的运行非 Google 版本的 Android 分支的设备就是 Amazon 的电子阅读器——Kindle。Amazon 也进入了手机（Amazon Fire Phone）和电视机顶盒（Fire TV）领域。Amazon 把它的 Android 分支称为 Fire OS，其变体对应的 Android 版本如下：

Fire OS 1	Gingerbread	2011
Fire OS 2	Ice Cream Sandwich	2012
Fire OS 3	Jelly Bean (4.2.2)	2013
Fire OS 4	KitKat (4.4.2)	2014
Fire OS 5	Lollipop	2015

放一些 Amazon 设备在你的实验室也许是有用的，因为 Amazon 确实有其独特的应用程序商店来向所有的 Fire 平板分发内容，这是应用的另一个市场。只要你不使用 Google 的具体服务，将 App 添加到 Amazon 的生态系统（包括 Amazon 网站）是一个很不错的想法。Amazon 应用商店的 App 也可以在黑莓设备上运行，因为黑莓手机有一个可以运行 Android App 的运行时。

2. 其他Android手机/平板电脑

最受欢迎的非 Google Android 设备是 Amazon 的设备。在美国，符合这种特征的设备包括 Barnes 和 Noble's Nook 平板。被微软收购之前，诺基亚有一个短暂的诺基亚 X AOSP 项目。

在美国以外，还有一些厂商成功营销 AOSP 设备，主要是印度和中国的制造商，他们生产廉价的手机。例如，中国 OEM 小米拥有全球 5.1% 的市场份额，并且 MIUI App Store 在短短一年多的时间内就实现了 10 亿的下载量。如果你的目标市场包括下个 10 亿的连接用户（提示：它应该可以达到），你也应该考虑测试这些设备。

2.6.5　其他选择

如果你难以维护一个设备实验室，仍然可以通过其他方式获取测试设备。

1. 远程设备测试

在线服务为你提供了通过网络接口访问实际设备的可能性。Testdroid、Appurify、Perfecto Mobile 和 Keynote 都是领先的供应商，拥有在线可用的移动测试设备。这些服务商管理和维护这些测试设备，你可以直接测试 App。这些服务通常有许多顶级的手机，并允许你运

行脚本或持续集成环境来测试 App。运行结果可以在浏览器中查看。虽然他们消除了维护本地设备的烦恼，但这些服务不太可能节省你的测试成本（事实上，它可能更贵）。这样做还有另一个缺点，没有实际的物理设备，你无法看到运行缓慢的情况或性能问题。这样你只是得到了测试结果，而不能真正看到 App 在这些设备上的运行状况。

> **Google 云测试实验室**
>
> 在 2015 年的 Google I/O 开发者大会上，Google 宣布了一项新的服务：网络物理设备。"几乎每一个品牌、每一个型号、每一个版本的物理设备，世界上的每一种语言、方向和网络状况下的虚拟设备。"提交的应用程序会在排名前 20 位的 Android 设备上免费进行自动测试，并报告结果和崩溃数据。截至 2015 夏天，这个工具已经临近发布了。

2. 开放设备实验室

如果你真的毫无测试设备的预算，或者因为不能在手中的设备上复现 bug 而困扰，可以尝试开放平台实验室（Open Device Lab，ODL，http://opendevicelab.com）。这些都是平民设备实验室（有些有永久的主页，有一些没有）（见图 2-1）。这些实验室设备的数量不同，但也许你能找到一些旧设备捐给附近的 ODL。如果你的社区没有 ODL，你也可以创建一个。你真正需要的是一些旧设备，以及与同行分享测试机的良好愿望。



图 2-1：世界各地开放的设备实验室

2.6.6 其他注意事项

当建立设备实验室时，你还需要维护一些其他的基础设施。Etsy 的 Lara Hogan 很好地讨论

了设备采集之外的设置实验室问题。[5] 其他要考虑的问题包括：

- 获得 USB 集线器以确保你的所有设备都能供上电
- 为你的移动设备建立一个专用 Wi-Fi 网络（以确保有足够的 Wi-Fi 吞吐量）
- 确保所有的设备在每次使用后都会擦除数据，而且不会意外升级系统
- 为每种设备准备合适的电缆和充电器

这些额外的细节对于准备设备实验室，以及开发人员开始测试是至关重要的。控制移动设备的软件也可以用来在设备实验室进行基本的综合测试。

2.6.7　我的设备实验室

2015 年，我和家人到欧洲旅行。因为我在旅行中也要工作，所以准备了一个便携式设备实验室，整个旅途中都带着。在工作中，我通常不在意屏幕大小，但我对 App 在旧版的 Android 操作系统上是如何工作的很感兴趣。我还经常测试被 root 过的设备。基于这些考虑，我会携带表 2-1 中所示的设备。

表2-1：便携式设备实验室

设备名称	操作系统	年　份
Samsung Galaxy Note II	Jelly Bean (rooted)	2012
Samsung Galaxy S4	KitKat (rooted)	2013
Motorola’s Moto G	Lollipop (rooted)	2013
Nexus 7	Lollipop	2013
Moto X(2014)	Lollipop	2014
Nexus 6	Marshmallow	2014
HTC One M9	Lollipop	2015
Samsung Galaxy Note 5	Lollipop	2015

这个列表中，Lollipop 占的比重非常大，但我想等 Marshmallow 更新的时候把它们升级到最新版本。这个列表覆盖了多个操作系统版本，设备发布时间也跨越了三年，操作系统版本覆盖了 85% 以上的用户，并且在未来的一段时间内也足够使用了。

2.7　总结

Android 设备数量众多，并会在新的领域中持续增长，如娱乐业和旅游业。没有理由认为 Android 的增长已经结束；你家里的所有东西可能都在运行 Android 系统，例如，在床上控制 Android 咖啡机。

注 5：Lara Hogan, “Etsy’s Device Lab,” Etsy - Code as Craft, August 9, 2013, https://codeascraft.com/2013/08/09/mobile-device-lab/.

为了确保 App 能够良好地在手机、平板电脑、汽车、电视、手表和太阳镜上运行，你可能需要专门的测试。适配的设备种类越多，测试的任务就越重（甚至可能是令人沮丧的）。通过获取实验室设备的参数（设置这些参数会让测试变得容易一些），你可以优化 App 性能，从而使客户更愉快。这将有助于增长用户数。在随后的章节中，我们将研究如何测试 App，以确保每个用户的设备性能都能达到最佳状态。

第 3 章

硬件性能和电池寿命

除了高性能的摄像头、更大更宽的屏幕、更小更快的处理器，电池容量也成为一部新设备十分重要的营销点。在评论新设备时，免不了要同上一代产品的电池续航能力进行比较。作为智能手机的使用者，我们不得不随身携带充电器——我们需要在家中、公司和车上给手机充电——来保证设备的电量。

设备都是不错的，问题在于你带着新买的设备离开商店后就会开始安装 App。雅虎在 2014 年的一项报告中指出，平均每个 Android 设备上会安装 95 个 App，但是日常用到的只有 35 个。[1] 这些 App 运行的时候会因使用不同的硬件功能而耗电。随着用户对这个问题关注度的提高，会有越来越多的工具能够帮助用户找出耗电量较大的 App。

本章会探讨 App 使用设备硬件的方式，以及优化这些交互的重要性——这能提升 App 的性能，令其更加流畅。此外，通过改善 App 与高级电池监控设备的交互，还能够减少 App 对电池寿命的影响。

3.1 Android的硬件特点

如今 Android 设备都配备了很多传感器，似乎没有什么是它们做不到的。然而，正如本叔叔对彼得 · 帕克（刚成为蜘蛛侠的时候）说的那样，“能力越大，责任越大”。Android 设备为开发者提供了很多很酷的工具，但如同每个木匠都会告诫你的一样，你必须小心对待这

注 1：Paul Sawers,“Android Users Have an Average of 95 Apps Installed on Their Phones, According to Yahoo Aviate Data,” The Next Web, August 26, 2014, http://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/.

些工具，否则它们可能会伤到你。虽然在软件开发中，身体不会受到真正的伤害，但是如果 App 没有做好对设备的适配工作，将会导致严重耗电、设备发热和其他不良影响，最终会影响用户。

我们手中的“能力”是十分惊人的，以 Samsung S5 的传感器为例：

- 指纹感应器
- 心律监控
- 环境光度感应器
- 温度湿度感应器
- 气压计
- NFC（近距离无线通信）
- 陀螺仪
- 加速计
- 蓝牙
- Wi-Fi
- 调频无线电
- 蜂窝无线电
- 前置、后置摄像头
- GPS
- 磁场探测器
- 光通量感应器
- 电池温度感应器
- 麦克风
- 触摸功能

那么，我们怎样才能快速地了解所有这些传感器的性能呢？最简单的方法就是观察耗电情况。设备中耗电量最多的部分，通常也是你最需要小心对待的部分。

3.2 少即是多

利用 Android 设备出色的特性，我们希望能给用户提供尽可能多的信息。但问题在于如果我们收集了过多的数据，就会影响设备的电池续航能力，因此我们必须在数据 / 信息和耗电之间找到一个合适的平衡点。进一步来讲，如果可以保证所有的任务都能够尽快地完成，也就能够保证性能和耗电之间可以实现良好的平衡。

Google 的一项报告指出，设备在活动状态下使用 1 秒的耗电量相当于待机 2 分钟的耗电量 ，这一点通过说明书来看更加有说服力：Nexus 5 声称它的待机（LTE，也就是 4G，和 Wi-Fi 开着，但并不使用设备）时长为 300 小时（12.5 天）。随着用户开始安装 App（或者

打开屏幕查看 App 消息)，电池续航时间会急速缩减至原来的 1/35（若在一般的 Wi-Fi 环境下使用，Nexus 5 仅承诺手机续航时间为 8.5 小时）！从整体来看，我们可以推测出，活动状态下，App 每 5 分钟就会消耗 1%~1.6% 的电量。同时 App 使用的资源越多，耗电量就越大。

上述情况在那些免费的、有广告的 Android 游戏中体现得尤为明显。有时，仅仅玩这类游戏 10~15 分钟，你就会发现手机背面热得发烫。这些 App 会在游戏使用 CPU、屏幕和其他硬件时下载大量广告。同时使用所有的这些组件会快速消耗手机电量，导致电池发热。2015 年 3 月发表的一项研究结果表明，相较于其他的 App 而言，带有广告的 App 会多使用 56% 的 CPU、22% 的内存和 15% 的电量。[2]

综上所述，我认为大部分耗电问题与硬件本身无关，而在于设计糟糕的 App 滥用设备功能。在本章中，我们将会讨论硬件使用中的一些错误做法，以及如何在 App 中避免上述问题（让 App 不再被打上“耗电量多”的标签）。

3.3 耗电原因

作为一位 Android 用户，你也许想知道经常使用的那些 App 对电池寿命的影响。通过研究手机中安装的 App，你可以发现一些耗电量较多的 App。通过学习以下的技术，你可以决定用户是否会在你的 App 中发现同样的性能问题。通过了解 Android 的耗电量评级方法，你可以保证你的 App 不会出现在这些报告中。你也许还能在自己的手机中发现一些耗电量大的 App（从而提升你个人设备的电池续航能力）。

3.3.1 Android能耗统计文件

正如本章接下来将讨论的，电池设置菜单会给出一份设备中所有 App 的耗电比例报告，这些耗电计算由 Android 能耗统计文件（部分）给出。在 Android 操作系统内部，有一个 XML 文件描述了系统中主要硬件组件的电量消耗情况。当 App 运行时（会唤醒设备不同的组件），系统即开始计算每个组件的电量使用情况，并将这部分电量消耗记录在你的进程名下。该 XML 文件同下面这份类似（电量消耗单位为毫安）：

```
<?xml version="1.0" encoding="utf-8"?>
<device name="Android">
    <item name="none">0</item>
    <item name="screen.on">65</item>
    <item name="screen.full">202</item>
    <item name="bluetooth.active">87</item>
```

注 2：Jiaping Gui , Stuart Mcilroy , Meiyappan Nagappan, and William G. J. Halfond,“Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers,”Proceedings of the 37th International Conference on Software Engineering (ICSE), May 2015, http://www-bcf.usc.edu/~halfond/papers/gui15icse.pdf.

```
<item name="bluetooth.on">1</item>
<item name="wifi.on">3</item>
<item name="wifi.active">240</item>
<item name="wifi.scan">129</item>
<item name="dsp.audio">29</item>
<item name="dsp.video">215</item>
<item name="radio.active">125</item>
<item name="radio.scanning">25</item>
<item name="gps.on">1</item>
<array name="radio.on">
    <value>4.5</value>
    <value>4.5</value>
</array>
<array name="cpu.speeds">
    <value>2457600</value>
    <value>2265600</value>
    <value>1958400</value>
    <value>1728000</value>
    <value>1574400</value>
    <value>1497600</value>
    <value>1267200</value>
    <value>1190400</value>
    <value>1036800</value>
    <value>960000</value>
    <value>883200</value>
    <value>729600</value>
    <value>652800</value>
    <value>422400</value>
    <value>300000</value>
</array>
<item name="cpu.idle">3.1</item>
<array name="cpu.active">
    <value>348</value>
    <value>313</value>
    <value>265</value>
    <value>232</value>
    <value>213</value>
    <value>203</value>
    <value>176</value>
    <value>132</value>
    <value>122</value>
    <value>114</value>
    <value>97</value>
    <value>92</value>
    <value>84</value>
    <value>74</value>
    <value>56</value>
</array>
<item name="battery.capacity">2800</item>
<array name="wifi.batchedscan">
    <value>.0002</value>
    <value>.002</value>
    <value>.02</value>
    <value>.2</value>
    <value>2</value>
```

```
    </array>
</device>
```

从上面的 XML 文件内容可以看出，当下移动设备中最耗电的硬件（不出所料）是屏幕、无线信号（蜂窝、Wi-Fi、蓝牙和 GPS）和 CPU（在高速处理下）。当我们回头评估 App 性能时，对性能和耗电产生影响的也是这些组件。所以，在优化 App 性能的同时，也可以提升用户设备的续航能力。

> **能耗统计文件**
>
> 能耗统计 XML 文件可以从 Android 系统级 APK 中找到。通过 Android 的文件资源管理器就可以找到它的位置。在 /System/Frameworks/ 目录下，复制一份 frameworks-res.apk 到你的工作电脑中，然后反编译就可以导出 /res/xml/power_profile.xml 文件了。
>
> 能耗统计文件中所有数值的单位都是毫安（mA）。让我们回忆一下高中物理课堂上所学的知识，毫安描述的是电流或者电荷量的大小，这个值越大，说明对应的部件耗电越快。电池的容量是用毫安时（mAh）来描述的，即 1 小时完全放电的电流大小。

3.3.2 屏幕

从能耗统计文件中可以看到，屏幕是耗电量最大的硬件之一（由上文中的能耗统计文件 `screen.full` 可知，当屏幕的亮度设置得较高的时候，电流可高达 202 毫安）。由于屏幕是 App 中必不可少的 UI 元素，而且通常 App 运行时都需要保持屏幕点亮状态，你也许会觉得无从掌控这方面的电能消耗。但事实是，在 UI 方面确实有一些方法可以减少屏幕的电能消耗。

一般来讲，Android 设备有两种主流的屏幕：发光二极管屏（LED）和液晶显示屏（LCD）。这些屏幕制造商都掌握着一些制造专利，例如源矩阵有机 LED（AMOLED）或者超级 LCD3，它们都拥有不同的视觉效果、能耗特性。但在这里，我们只需要分成两种屏幕类型进行分析即可。

1. LCD

简单来讲，LCD 屏幕中包含了成千上万个液晶分子来负责每个像素点的颜色显示，并通过一个背光来将它们同时照亮。这种显示方式的能量消耗主要在照亮液晶分子的背光上，而不是像素的颜色显示上，也就是说，每一个像素消耗的能量是相同的，和它们呈现的颜色无关。

2. LED

对于 LED 屏幕，每个像素点同时发出颜色和光亮。每个像素点是由红、蓝、绿三种颜色

的 LED 所组成的（这些构成方式十分复杂，而且显示屏狂热者还在孜孜不倦地辩论怎样才是最好的构成）。通过控制每个 LED 的亮度和颜色，像素点就能呈现出需要显示的颜色。由于每个像素点是由三种光源共同呈现的，所以不同颜色的能量消耗也大相径庭。黑色不使用任何颜色，所以不消耗能量，而与此相对的白色，使用了所有颜色和最高亮度的光，所以会消耗更多的能量。由此可知，较暗的颜色会比较亮的颜色更节能，这也是一些新闻和社交 App（那些有很多留白的 App）使用黑色背景的原因。

虽然使用暗色背景在 LCD 屏幕中收效甚微，但 LED 屏幕潜在的降低耗电量的可能性，足以让我们考虑选择暗色背景。

在第 4 章我们将对屏幕性能和 UI 性能做更深入的讨论——并不局限于电源和能耗的分析。

3.3.3　无线设备

从能耗统计文件中可以看到，蜂窝无线和 Wi-Fi 使用的电量相近。在第 7 章中，我们将讨论 Wi-Fi 和蜂窝无线在连接方式上的不同。总的来说，蜂窝无线和 Wi-Fi 最大的不同点在于，蜂窝无线的持续时间要比 Wi-Fi 长很多，使用蜂窝无线的会话时间也会变长，结果便是比 Wi-Fi 消耗更多的电能。从根本上讲，若 App 要使用无线传输，最好的性能提升方式是一次下载尽可能多的数据，然后关闭无线设备。减少请求次数是一个一举两得的办法，不仅可以提升屏幕的加载速度，也可以节省电量（我们会在第 7 章中讨论更多的细节）。

另一个无线接收设备是 GPS（通常会被忽略）。当使用定位服务时，精准地了解你所需要的位置信息对于节省时间（和电能）是非常重要的。如果你只需要一个粗略的位置信息，那么蜂窝无线就可以提供足够的数据而无需使用 GPS。因为信号基站的位置会存在设备本地，所以如果用户不移动，甚至可以不使用蜂窝无线连接。避免使用 GPS 可以加快 App 的响应速度（设备的位置服务依然可用），同时节省电能。

> **GPS 失效备援**
>
> 请不要忘记，当用户处于地下室时，设备可能接收不到 GPS 信号。如果在一段时间内你都接收不到 GPS 信号，那么请确保关闭 GPS。
>
> Android 的 App 中做不到这一点的比比皆是，常常开着 GPS 长达 40 分钟却不用。这种行为不仅不能节约电量，而且也丝毫不能提升用户体验。

在第 7 章，我们将对网络性能的细节作更深入的探讨。

3.3.4　CPU

游戏或者是含有大量计算的 App 都会使 CPU 的占用率急剧提高。此外，如果 App 有大

量的后台计算要做，CPU 就会被唤醒来进行额外的计算工作。正如能耗统计文件显示的，CPU 的占用率越高，电量消耗得就越快。

CPU 的占用率和屏幕、网络以及设备上进行的所有计算息息相关，因此整本书都会涉及 CPU 占用率的优化问题。

3.3.5 其他传感器

能耗统计文件列出了 Android 设备的主要组件。此外，正如我们在第 1 章所介绍的一样，手机上有很多开发者可以用来当作 App 亮点的其他传感器。

注册了一个传感器之后，就可以使用 `getPower()` 方法来获取传感器消耗的电量了。Google Play 上有很多免费的 App 可以列举出设备上的传感器的使用情况（以毫安为单位）。下面列出了 Nexus 6 上的传感器的电量消耗情况：

加速传感器	0.3 毫安
磁力传感器	10 毫安
陀螺仪传感器	1.5 毫安
距离传感器	12.675 毫安
光线感应传感器	0.175 毫安
气压计	0.004 毫安
旋转矢量传感器	11.8 毫安
地磁旋转矢量传感器	10.3 毫安
方向传感器	11.8 毫安
线性加速传感器	1.8 毫安
重力传感器	1.8 毫安
倾斜传感器	0.3 毫安
设备位置分类器	5.6e-45 毫安
步行检测传感器	0.3 毫安

每种传感器都有确定的最大信号频率。作为开发人员，设置合适的采样频率很重要。除了传感器，处理这些信号数据也会占用设备的 CPU 和内存，过度的采样会浪费资源。

Android 内置了很多可用的行业标准样本率（SENSOR_DELAY_NORMAL、SENSOR_DELAY_GAME，等等）。

最后，一定要确保在调用完传感器之后将其注销。如果回调监听者保持活跃，传感器将继续报告数据，这势必会造成不必要的处理器负载、内存占用和电量消耗。

海森堡的测不准原理

一本 Android 书中也有量子力学？维尔纳 · 海森堡的研究表明，观察世界必然干扰世界。监测设备在监测电量使用情况的同时也在消耗电量，从而对监测结果产生干扰。不过很多可以连接的外部工具在监测期间并不会干扰监测结果，例如我使用的能耗追踪计（本书还会介绍几种手机能耗追踪工具）。

3.3.6 休眠

App 不工作的时候让其休眠是很重要的。释放传感器和信号发射器并允许屏幕休眠，会节省很多电量。让 App 休眠至关重要，算准时机激活 App 同样很重要。合适的唤醒频率可能会极大地延长电池的使用时间。

3.3.7 WakeLock和Alarm

根据以往的开发经验，开发者通常使用 WakeLock 和 Alarm 来唤醒设备处理信息。在与用户无交互的情况下，开发者可以通过在 App 中实例化一个 WakeLock 和 Alarm 来唤醒设备处理信息。WakeLock 也可以用来阻止设备恢复休眠状态。既然我们已经了解了 Android 设备每种硬件的耗电量，你应该知道在后台唤醒设备会对电池性能造成多么严重的影响。此外，当你的 App 唤醒了设备，也就为其他 App 打开了工作的大门时，它们有可能还会打开信号发射器。在 Lollipop 系统中，Android 增加了更加智能的同步唤醒功能（JobScheduler，3.6 节中会深入地介绍该 API）。

1. WakeLock

WakeLock 可以唤醒（保持唤醒状态）移动设备的部分组件。使用得当，这就是一个有用的 App 特性。我记得有一款没有使用 WakeLock 机制的赛车 App，在赛车途中屏幕会变暗休眠，结果车就撞墙了。不用多说，我卸载了这个游戏！屏幕 WakeLock 就是电影流媒体 App 在播放电影期间保持屏幕点亮，或者是音乐流媒体 App 在播放期间保证音频频道正常运行，而其他不需要的设备组件休眠的机制。对于某些类型的 App，WakeLock 是很重要的用户体验。然而，如果处理不当，WakeLock 也会导致极端的耗电。

如果要强调一下，那么 WakeLock 是电源管理 API 的一部分。对于这个类的注释描述如下所示：

这个类能够帮助开发者控制设备的电源状态。这个 API 的使用会影响设备的电源寿命。除非真的需要，否则不要获取 PowerManager.WakeLocks，要尽量少用 WakeLock 并确保使用之后尽快释放。[Android 中强调]

WakeLock 用于保持屏幕为点亮状态的时候和传感器一样，只要屏幕休眠就要确保 WakeLock 被释放。

WakeLock 检测

搭载 KitKat 之前的系统的设备，Google Play 上有很多免费的 WakeLock 检测 App，可以用来进行 WakeLock 问题诊断。这些 App 做的分析工作基本相同，选择一款使用即可。在 KitKat 和之后的系统版本中，系统提供了 WakeLock 的检测 API（`adb shell dumpsys batterystats` 命令的一部分），但是只对 root 过的设备上的 App 可用。本书后面会介绍一些电池性能分析工具的内部工具，以及如何使用这些工具来诊断 App。

2. Alarm

Alarm 允许开发者设置时间执行特定的操作，特别是 App 在后台运行或者是设备处于休眠状态的时候。例如，每小时唤醒设备一次并且检查服务器更新。虽然这确实也是一种更新 App 的方法，但它有副作用，就如 Android SDK 的警告一样："设计糟糕的 Alarm 会导致电量消耗和服务器的重负荷。"

我就曾见过一些流行的 App 利用 Alarm 来进行数据同步。例如，一款 App 每 3 分钟激活手机一次，建立网络连接以更新消息，这样每天就会产生 480 次额外的连接（假设手机的电池能持续 24 小时），仅仅后台运行就会消耗 10%~20% 的电量。

有时间精确性要求的提醒应该设置一个准确的 Alarm（例如，创建一个闹钟 App）。其他的情况下可以使用不精确的 Alarm，操作系统会自动协调、合并 Alarm 来节省电量。下面的代码会每天在 alarmTime（操作系统自动协调 Alarm 的时间，但是不确定是 24 小时的什么时候）的时候唤醒一次设备：

```
alarmManager.ssetInexactRepeating(AlarmManager.RTC_WAKEUP,alarmTime,
        AlarmManager.INTERVAL_DAY, alarmIntent);
```

3.3.8 Doze模式

如上文所说，设备被唤醒的次数越多，设备的电量消耗得就越快。闲置状态下，App 每使用一次 WakeLock 和 Alarm 都会加速设备的电量消耗。研究表明，70% 的电量消耗都是由设备闲置时开启网络连接以更新数据引起的。本书将在第 7 章中讲解网络连接的相关优化，但是可以肯定地说，对 App 唤醒设备的次数进行限制必然会节省更多的电量。

2015 年 Android 发布了 Marshmallow，Google 官方加入了 Doze 模式用来限制设备被唤醒的次数。这样做是以 App 的“数据更新”（data freshness）为代价的，但是如果电池没电了，得到的数据再实时又有什么用呢。设备允许特定的 App 更新（当然，屏幕点亮的时候，所有的 App 都可以更新数据）。

那么 Doze 模式是怎么工作的呢？该模式有以下几种状态。

- *激活状态*
  屏幕点亮。
- *休眠状态*
  屏幕休眠，但是设备处于唤醒状态。
- *待闲置状态*
  准备进入闲置状态。
- *闲置状态*
  设备休眠。
- *闲置维护状态*
  队列中 Alarm 和更新任务的短暂的执行机会。

强制设备进入 Marshmallow 的这几种不同的状态的命令如下：

```
adb shell dumpsys battery unplug //设备停止充电的命令
adb shell dumpsys deviceidle step //重复该命令,遍历各种状态
```

实际上，设备在灭屏的状态下从休眠状态到待闲置状态需要 30 分钟，并且需要再经过 30 分钟才会进入闲置模式。一旦进入了闲置模式，设备将推迟所有的 Alarm 直到下一个闲置维护期（60 分钟后）。闲置维护期之间的延时是不断增加的（1 小时、2 小时、4 小时、6 小时），最大的间隔是 6 小时。所有的 Alarm 和 WakeLock 都会被暂停直到维护期的到来，对于那些长期处于闲置状态的设备（像平板电脑），这无疑会节省大量的电量。

开发者应该在 Doze 模式下测试一下开发的 App，以确保 Doze 模式下有多个通知的时候只提示一条消息。

3.4 基本的电量消耗分析

我们已经讨论过了硬件如何消耗电量、Android 如何计算 App 消耗的电量，以及低效的唤醒如何导致高耗电量。如果说 App 是消耗电量的原因，那怎么确定设备上最耗电的 App 呢？电量的设置菜单详细地展示了所有移动 App 的能耗信息，更重要的是，所有的 Android 用户都可以访问。当务之急是不要让你的 App 出现在这些诊断名单之中，因为用

户很可能根据这个名单选择卸载你的应用。

如图 3-1 所示，当第一次打开电量菜单的时候（设置→ 电量），你会看到一张耗电量的图标（是从 100% 的电量算起的）。下面这张图表展示了特定时间段内所有 App 的耗电量。下面来分析一下这张图表能告诉开发者（还有用户）什么。

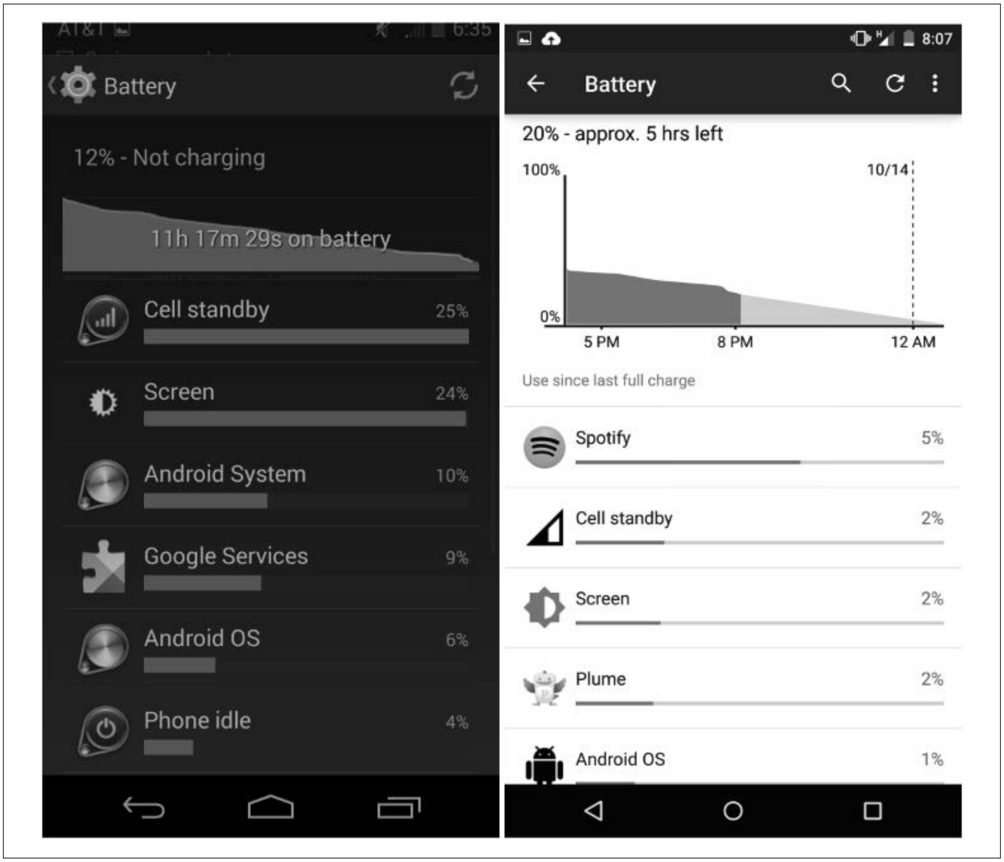


图 3-1：KitKat（左）和 Lollipop（右）的电量菜单

从上图中可以看到从 KitKat 到 Lollipop 的电量菜单的优化。KitKat 显示了当前电量的使用情况，Lollipop 除此之外还增加了剩余电池寿命的预测（基于当前的使用情况）。通过触摸顶部的图，还可以在扩展菜单中看到更详细的电量使用情况（见图 3-2）。

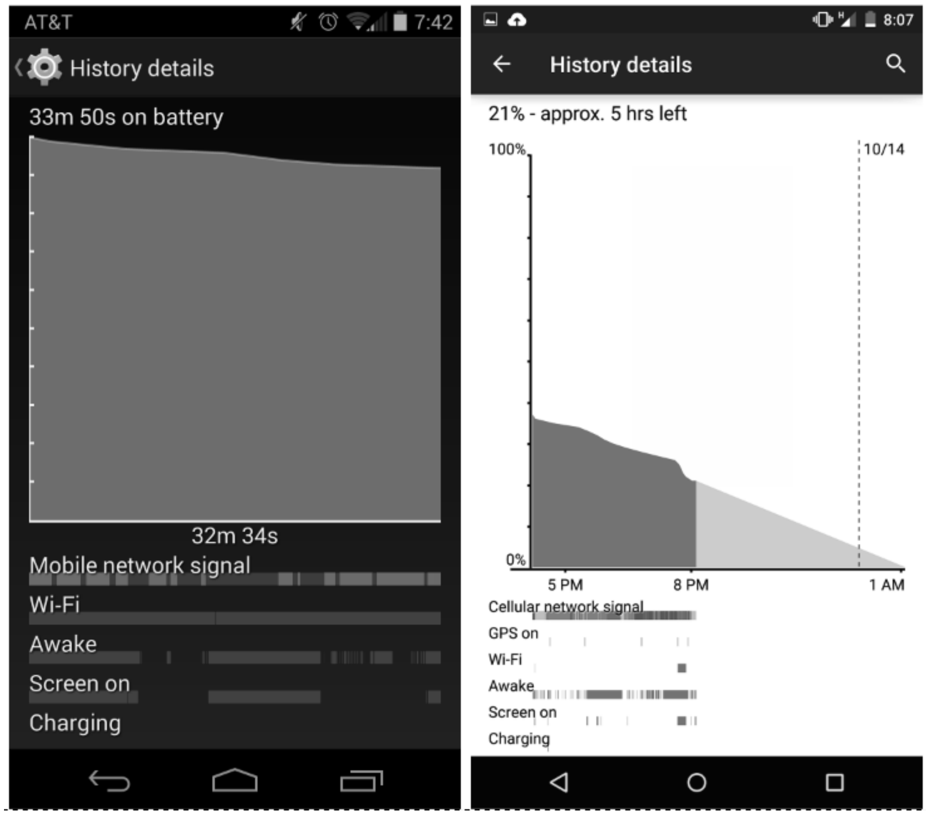


图 3-2：KitKat（左）和 Lollipop（右）的能耗详情（另见彩插）

扩展菜单显示了设备在各种网络状态（绿色 / 黄色 / 红色指示信号质量）下的连接时间、Wi-Fi 活跃的时间、设备唤醒时间、屏幕点亮的时间，以及设备的充电时间。用户应该更偏爱 Lollipop 的视图，因为它不仅展示了电量的真实数据（绿色），而且展示了预测的剩余时间（灰色）。开发者需要查看设备和 App 的性能，应该会更喜欢 KitKat 的视图，因为屏幕上只显示设备的真实电量的使用情况，更方便阅读。

从 KitKat 的图中可以看出，在亮屏手机被唤醒而网络情况较差的时间段内，电池在快速地放电（左）。Lollipop 的设备上在截图之前也存在一段类似的下降（图中由绿变灰的过程）。

作为一名开发者（和用户），要注意 App 一个重要的能耗指标——灭屏时唤醒设备。这表示，在用户未使用设备的状态下，App 利用 WakeLock 或者 Alarm 使用了设备。如果这种情况频繁发生，就需要查看 App 的耗电量，并找出是哪些 App 造成的这个问题。

3.4.1 详细的App电量消耗分析

返回到电量的主菜单，滑动到下面会看到电量的图标数据，里面有每一个 App 能耗的相关分析。根据我的经验，这些百分比都不会非常高，这可能是因为每个 App 只消耗了非常小的电量。通过选择菜单中具体的 App，可以看到前台 App 的 CPU 占用率和总的 CPU 占用率（前台和后台的总和）。此外，菜单还提供了数据流量的使用情况（前台 / 后台，蜂窝无线网 /Wi-Fi）和 App 保持设备为唤醒状态的时间。前台 App 的占用率和数据量是很高的（耶！人们在用你的 App），但是很高的后台占用率就意味着 App 可能正在将设备从休眠状态中唤醒。

例如，图 3-3 中显示了 Facebook 和 Spotify（在 KitKat 系统中）的详细耗电量数据。

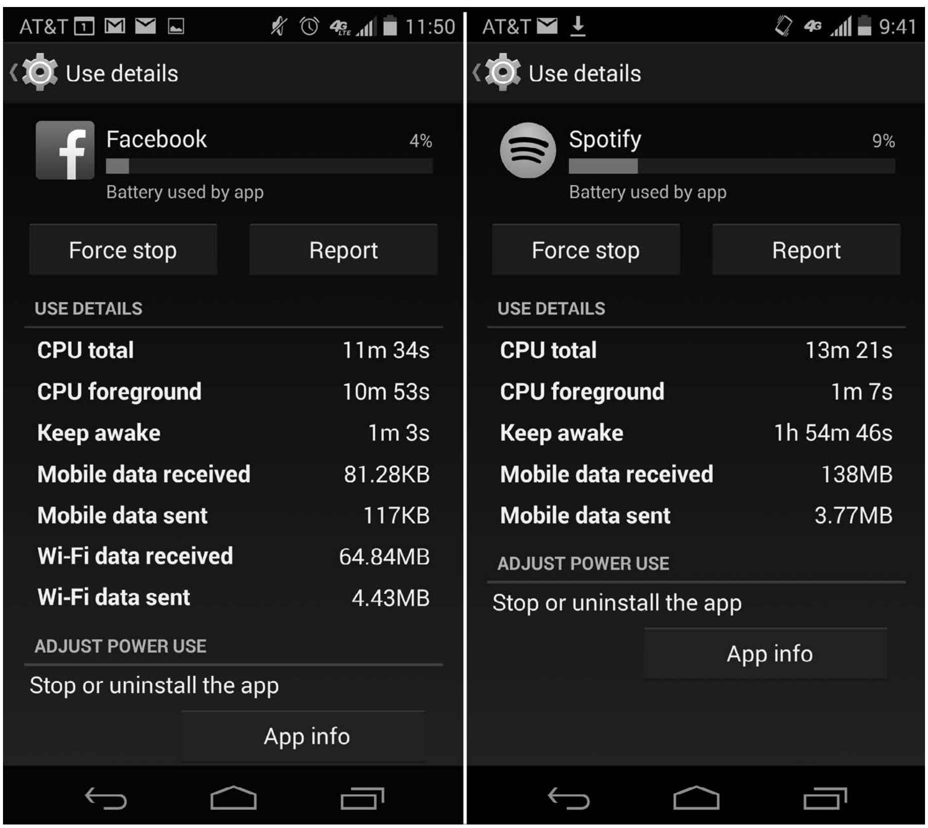


图 3-3：Facebook 和 Spotify 的能耗详情

在这个视图中，Facebook 被认为（罪魁祸首？）占用了设备 4% 的能耗（按照 3.3.1 节中的能耗统计文件计算）。Facebook 的 CPU 占用主要在前台（11.5 分钟的占用中有 11 分钟在前台），另外 30 秒的 CPU 占用在后台，可能和更新服务器数据有关。在新闻推送中看 1

分钟视频，App 就设置屏幕 WakeLock 保持唤醒手机 1 分钟，防止灭屏。Facebook 并没有使用大量的蜂窝网数据，但是 Wi-Fi 数据量很惊人（但是并不意外：除了电影，还有大量的图片下载）。

Spotify（一款音乐播放 App）的能耗情况和 Facebook 明显不同。播放音乐的大多数时候，我的手机都处于灭屏状态（手机是放在兜里面的）。电量表格可以证明，大多数的 CPU 处理都发生在后台（大约 12 分钟），并且设备持续唤醒状态（可能是音频 WakeLock 的作用）至少 2 个小时。数据流量很高，但对于播放 2 个小时的流媒体音乐来说不算过高（如果没有观察这些 App 的经验，很难知道这个）。

电量菜单中的数据流量使用信息是自上次充电（当电池数据被重置为自动）之后的发送和接收到的数据量。然而这个数据量并不能说明数据传输的效率问题（在我看来，这才是开发者想要在电量菜单中看到的）。很明显，Google 在 Lollipop 的电量菜单中改变了数据量的报告方式，如图 3-4 所示。

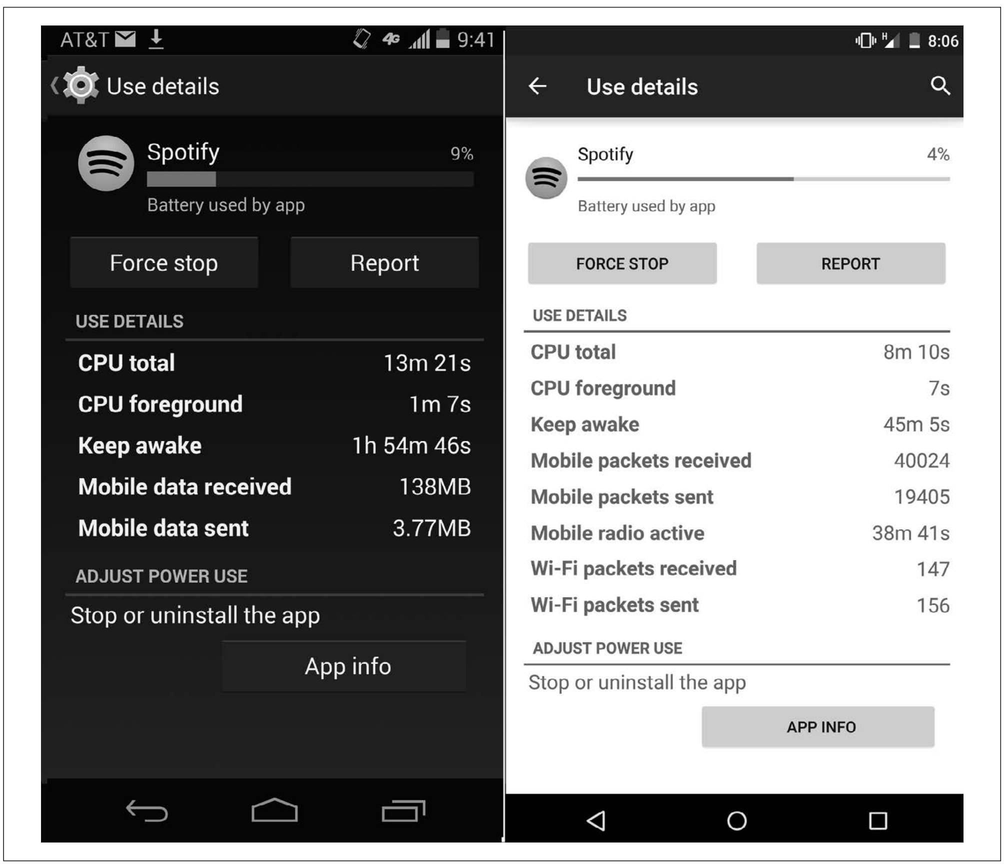


图 3-4：Spotify 在 KitKat（左）和 Lollipop（右）中的能耗详情

注意，在 Lollipop 的电量菜单中数据流量的单位是包而不是 KB，并且还将数据量细分为 Wi-Fi 数据量和蜂窝无线网数据量两种，而且还显示了手机信号发射器的使用时间。基于这些给力的新报告，我们可以估算出信号流量有多密集（密集到 39 分钟内发送了 40 000 个包，每 60 毫秒接收一个包）。密集的信号流量意味着要在最小的信号交互时间内尽可能快地发送数据并且允许用户消费数据。

3.4.2　能耗数据和数据流量

为了更好地处理移动 App 的数据流量问题，你可以使用数据流量菜单（注意菜单里记录的只有蜂窝无线网的数据流量使用情况，Wi-Fi 通常是不计费的）。选中一款 App，就可以看到该 App 的前台和后台的数据流量的使用情况。在图 3-5 的屏幕截图中，移动滑块只显示两天的数据流量使用情况，但是我们只看到了 24 小时的精确的前台和后台的数据流量使用情况。

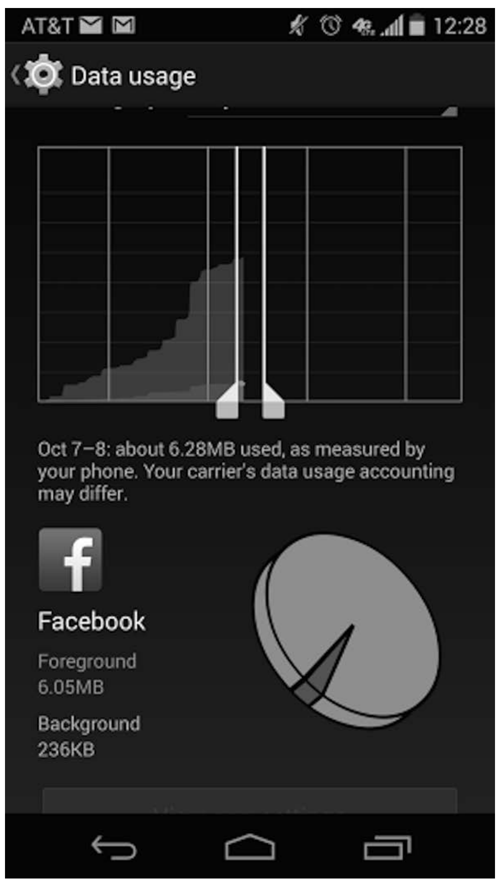


图 3-5：Facebook 在 KitKat 的数据流量使用详情

在 Lollipop 中，这个菜单也有所改变，菜单去掉了允许用户改变计量日期的滑块。如图 3-6 所示，如果想要通过菜单的数据来分析问题，就需要在每次测试之前重置数据流量，以确保只展示测试期间的数据流量的使用情况。

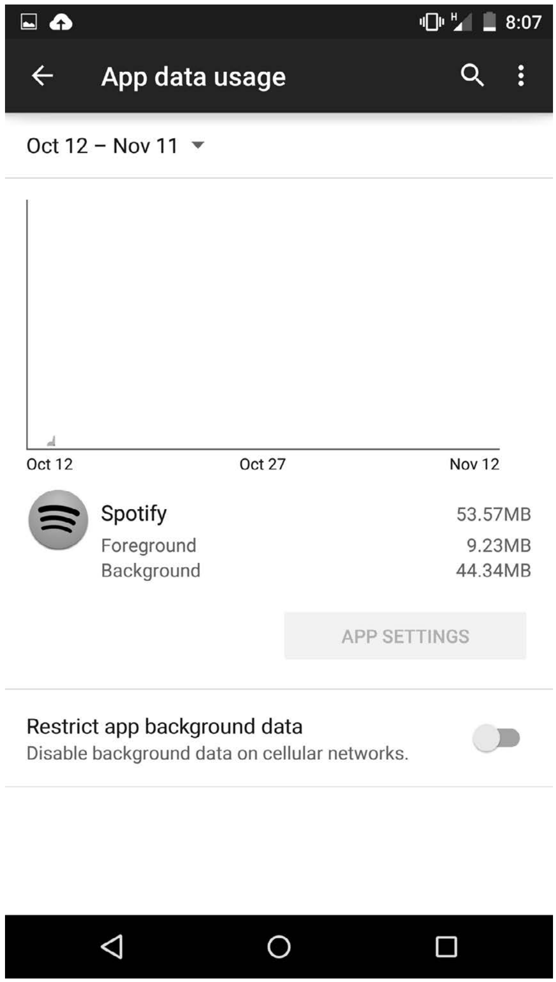


图 3-6：Spotify 在 Lollipop 的数据流量详情

生成图 3-6 之前，我已经重置了蜂窝无线网的数据流量，这样我就可以对流量和数据包数进行比较了。比较一下这里的数据流量和图 3-4 中的数据包，我们现在知道接收到 40 024 个数据包（从电量菜单可以看出）耗费了 53.57MB 的流量，有效值大约是 1403 字节 / 包，23.6KB/s。这是一种很密集的数据流量模式（用于流媒体音乐）。如果手机上存在数据有效转化率和吞吐量较低的 App，你可以考虑禁用这些 App 的数据（或者可能是后台数据）。这些 App 可能导致信号发射器降低效率并增加额外的电量消耗。

细致的电量监测有助于发现过度消耗流量的移动 App，然后基于收集到的这些数据，你可以决定保留或卸载 App。结合这些菜单的信息，你就可以看出很多移动 App 的电量消耗和流量消耗的问题。今天，我们想要确定 App 的能耗情况还需要结合这两种菜单的数据。但这只是时间问题，未来这些比较工具会变得更简单、更主流化。

3.4.3 App休眠

在 Marshmallow 系统，Google 宣布了一个新的功能叫 App 休眠。App 休眠会阻止那些不常用的 App（几天没有用过的 App）连接网络或者是运行任何程序直至设备充电。对于用户而言，这就意味着不经常使用的 App 不会消耗电量，从而延长电池的续航时间。

用 `adb shell dumpsys usage stats` 命令可以查看 App 的进程和它们最后活跃的时间，列出 App 最后一天 / 星期 / 月 / 年的使用情况。在设置的开发者选项中有一项新的“未启用 App”选项，该选项中列出了 App 以及 App 当前是启用还是未启用的状态（处于 App 休眠的状态）。

3.5 高级电池监控

我最初的监控电池使用情况的测试只是使用了各种 Android 的设置菜单项。这样的测试可以从宏观层面查出一些高耗能的地方，有效地发现设备上耗电较多的 App。然而，从开发人员的角度来看，它还有很多需要改进的地方。在 KitKat 版本中，Android 增加了电能统计系统转储（这就是 Google Play 中 WakeLock 监控应用全部停止工作的原因）。到了 Lollipop 之后，它又提供了更多的信息，并且添加了一些可视化工具。

3.5.1 电能统计

电能统计（`batterystats`）包含大量描述设备（包括所有正在运行的进程）电池使用情况的数据。比起刚开始的 KitKat 版本，Lollipop 中提供了更详细的数据（包括每次 WakeLock 动作的记录）。收集跟踪数据之前，最好重置一下数据。为了收集尽可能多的数据，你可以开启报告全部 WakeLock 的功能（这个功能只支持 Lollipop 及以上的系统）：

```
adb shell dumpsys batterystats --reset

adb shell dumpsys batterystats --enable full-wake-history
```

注意，重置了电能统计也会重置电池设置选项的全部数据。

为了让大家更好地理解电能统计系统转储，我们可以在命令行界面开启电能统计转储（在这里，下载上次完全充电后的所有电能数据）：

```
adb shell dumpsys batterystats --charged
```

你可以看到终端里打印出了很多数据，但这些信息是什么意思呢？让我们看一下输出信息中一些比较有用的部分。设备的一些基本状态信息——设备在不同无线状态下的时间、发送的数据量、设备保持全部或部分唤醒的时间等。

下面的测试数据来自一段 30 分钟的电能统计转储。这段时间内，我玩了一会儿游戏（还收到了一条 Facebook 的消息），然后让手机闲置。第一张表显示，电池每 2 分钟损失 1% 的电量。这个表的顺序是自下而上的（开始时的电量为 97%，结束时电量为 88%）。电量消耗的速率是非常稳定的，但是可以看到，设备在闲置和使用状态下是不同的。

```
放电时间：
  #0: +2m28s313ms to 88 (screen-on, power-save-off)
  #1: +2m38s364ms to 89 (screen-on, power-save-off)
  #2: +2m27s323ms to 90 (screen-on, power-save-off)
  #3: +2m8s449ms to 91 (screen-on, power-save-off)
  #4: +2m17s115ms to 92 (screen-on, power-save-off)
  #5: +2m7s924ms to 93 (screen-on, power-save-off)
  #6: +2m17s693ms to 94 (screen-on, power-save-off)
  #7: +2m6s425ms to 95 (screen-on, power-save-off)
  #8: +1m50s298ms to 96 (screen-on, power-save-off)
  #9: +3m0s436ms to 97 (screen-on, power-save-off)
```

下面这张表包含了设备统计信息。我们可以看到：打电话使用电池刚超过了 30 分钟，其中 3.5 分钟处于屏幕关闭状态，但是关闭屏幕后设备仍然保持了 52 秒唤醒状态。27 分钟处于屏幕开启状态，亮度设置为暗（背景为暗，亮度约为 40%）。蜂窝网络一直在无和良好之间波动，但大多时间处于低到中。Wi-Fi 信号强度是 4 级，但是 Wi-Fi 是处于关闭状态：

```
统计消耗：
  System starts: 0, currently on battery: false
  Time on battery: 30m 36s 621ms (99.3%) realtime,27m 58s 456ms (90.8%) uptime
  Time on battery screen off: 3m 31s 100ms (11.4%) realtime, 52s 935ms (2.9%) up
  Total run time: 30m 48s 839ms realtime, 28m 10s 674ms uptime
  Start clock time: 2014-10-17-22-54-33
  Screen on: 27m 5s 521ms (88.5%) 1x, Interactive: 27m 5s 837ms (88.5%)
  Screen brightnesses:
    dark 27m 5s 521ms (100.0%)
  Total full WakeLock time: 29m 16s 938ms
  Total partial WakeLock time: 17s 153ms
  Mobile total received:187.99KB, sent:201.15KB (packets received 750, sent 742)
  Phone signal levels:
    none 35s 29ms (1.9%) 10x
    poor 11m 7s 494ms (36.3%) 96x
    moderate 18m 29s 647ms (60.4%) 94x
```

```
    good 24s 451ms (1.3%) 7x
  Signal scanning time: 0ms
  Radio types:
    hspa 15m 12s 768ms (49.7%) 49x
    hspap 15m 23s 853ms (50.3%) 49x
  Mobile radio active time: 14m 32s 106ms (47.5%) 41x
  Mobile radio active unknown time: 1m 23s 222ms (4.5%) 21x
  Mobile radio active adjusted time: 0ms (0.0%)
  Wi-Fi total received: 0B, sent: 0B (packets received 0, sent 0)
  Wifi on: 0ms (0.0%), Wifi running: 0ms (0.0%)
  Wifi states: (no activity)
  Wifi supplicant states:
    disconn 30m 36s 621ms (100.0%) 0x
  Wifi signal levels:
    level(4) 30m 36s 621ms (100.0%) 0x
  Bluetooth on: 0ms (0.0%)
  Bluetooth states: (no activity)
```

下面的“Device battery use since last full charge”（上次完全充电后的电池使用情况）表格按百分比显示了这段时间内电池的使用情况。这张表的内容显得比较简单，因为电池的消耗较为稳定。我看到这个表中的结果有几个百分点的差异。最后一行显示出屏幕关闭时消耗的电量，这可以作为后台有应用运行的标识：

```
Device battery use since last full charge
  Amount discharged (lower bound): 10
  Amount discharged (upper bound): 11
  Amount discharged while screen on: 11
  Amount discharged while screen off: 0：
```

最后一个表我只粘贴了一部分，因为原始的表格太长了。它显示了所有消耗电量的进程，以及该进程占总耗电量的比例：

```
  估计电量使用(mAh)：
    Capacity: 3220, Computed drain: 359, actual drain: 322-354
    Uid u0a117: 106
    Screen: 96.6
    Uid 1000: 26.1
    Uid 0: 24.9
    Cell standby: 22.9
...
```

在设备数据之后是具体 App 的数据。首先是一个显示每个进程无线网络使用情况的列表。每一个进程都有一条数据，表示每个包发送 / 接受所消耗的时间（mspp——数据包到达的频率，这代表了效率）。对于高效的数据传输，mspp 应该尽可能低。列表还包括了数据包计数、无线网络使用时间和进程开启无线网络的次数。在报告的其他部分，可以看到几张将 Uid 解析为可读的 App 名称的表格（我在这里隐藏了这些进程的名字）。

```
  手机上每个App每个包的ms：
    Uid u0a111: 1569 (116 packets over 3m 1s 969ms) 26x
    Uid u0a77: 851 (119 packets over 1m 41s 309ms) 6x
```

```
    Uid u0a117: 592 (30 packets over 17s 772ms) 2x
    Uid u0a96: 541 (178 packets over 1m 36s 266ms) 9x
    Uid u0a116: 531 (106 packets over 56s 234ms) 5x
    Uid u0a102: 420 (248 packets over 1m 44s 152ms) 8x
    Uid u0a73: 361 (33 packets over 11s 906ms) 2x
    Uid 0: 339 (113 packets over 38s 347ms) 14x
    Uid u0a10: 335 (389 packets over 2m 10s 380ms) 14x
    Uid u0a28: 239 (160 packets over 38s 221ms) 5x
    TOTAL TIME: 12m 56s 556ms (0.0%)
```

最后，对于每一个进程，电能统计都会列出所有的 WakeLock，以及每个 App 的所有数据、WakeLock、电量消耗的清单。在这里我只展示了一个 App（u0a116，Facebook Messenger）。

```
  u0a116:
    Mobile network: 6.49KB received, 5.94KB sent (packets 63 received, 43 sent)
    Mobile radio active: 56s 234ms (6.4%) 5x @ 531 mspp
    Wake lock *vibrator* realtime
    Wake lock AudioMix realtime
    Wake lock *alarm*: 26ms partial (3 times) realtim
    Wake lock   wake:com.facebook.orca/com.facebook.push.mqtt.receiver.MqttReceiver
    TOTAL wake: 26ms partial realtime
    Vibrator: 100ms realtime (1 times)
    Foreground for: 1m 10s 792ms
    Active for: 30m 36s 621ms
    Proc com.facebook.orca:
      CPU: 1s 160ms usr + 470ms krn ; 0ms fg
    Apk com.facebook.orca:
      6 wakeup alarms
      Service com.facebook.push.mqtt.receiver.MqttReceiver:
      Created for: 148ms uptime
      Starts: 7, launches: 7
    Service com.facebook.conditionalworker.ConditionalWorkerService:
      Created for: 61ms uptime
      Starts: 1, launches: 1
    Service com.facebook.analytics.service.AnalyticsService:
      Created for: 1m 16s 407ms uptime
      Starts: 2, launches: 2
    Service com.facebook.orca.chatheads.service.ChatHeadService:
      Created for: 1m 11s 176ms uptime
      Starts: 1, launches: 1
    Service com.facebook.push.fbpushdata.FbPushDataHandlerService:
      Created for: 52ms uptime
      Starts: 2, launches: 2
    Service com.facebook.orca.notify.MessagesNotificationService:
      Created for: 540ms uptime
      Starts: 4, launches: 4
```

当我正在记录跟踪数据的时候，我妻子发来了一条 Facebook 消息。这张表给出了大量的信息，体现了 Facebook Messenger 在接收信息的简单过程中做了哪些事情：

- 蜂窝无线网络开启了大约 56 秒，接收 6.49KB，发送 6KB

- 手机使用 WakeLock 发出了振动来通知我
- 手机使用音频 WakeLock 发出通知
- 这些通知 partial WakeLock 花费了 26 毫秒
- 振动持续了 100 毫秒，但它不依赖于 WakeLock

Facebook Messenger 一直在后台运行，Battery Historian（电池分析工具）显示，虽然 Facebook Messenger 有 30 分 36 秒处于活跃状态，但只有 1 分 10 秒是在前台，CPU 占用时间也只有 160+470=630 毫秒。简单地讲，应用程序一直在等待一条消息到达，当消息到达的时候，它唤醒了设备 1 分钟用来提示我。

这 1 分钟，主要是被 `ChatHeadService` 和 `Analytics  Service` 占用。`ChatHead` 在我设备的前台开启一个通知窗口，表示我收到了一条消息。它活跃了 1 分钟，等待我回复一条消息。当 `ChatHeadService` 结束之后，`AnalyticService` 又启动了几秒，向服务器汇报——事实上，我并没有做出回复。

3.5.2 Battery Historian

`Batterystats` 可以在很大程度上帮助我们确定 App 消耗电量的原因。它有大量的、有用的、可深入研究的细节，可以用来了解 App 的表现以及潜在的问题。然而，在那么长的文本中找到有用的信息就像大海捞针。为了简化分析，Google 开发出了 Battery Historian（http://github.com/google/battery-historian），它将原始的 `batterystats` 输出文件以图形化的方式生成为一份 HTML 文档。运行下面的命令，就可以根据 `batterystats` 创建一个可视化的网页：

```
adb bugreport > bugreport.txt  //download the output to your computer
./historian.py  bugreport.txt > out.html //create the html file
```

在 2015 年的 Google 开发者大会上，Battery Historian 2.0 正式发布了。新的 Battery Historian 完全用 Go 语言重写，并且提供更全面的信息，能够帮助你深入研究 App 的能耗数据（注意，设备必须运行 Lollipop 或更新版本的操作系统）。首先，来看一看 Battery Historian 的两个版本共同包含的部分（在版本 2.0 中标记为 Historian-Legacy）。

我们将继续评估相同的数据追踪，但现在是在浏览器中。在图 3-7 中，我们可以清楚地看到 Battery Historian 的图表，它包含了设备记录的大量数据。

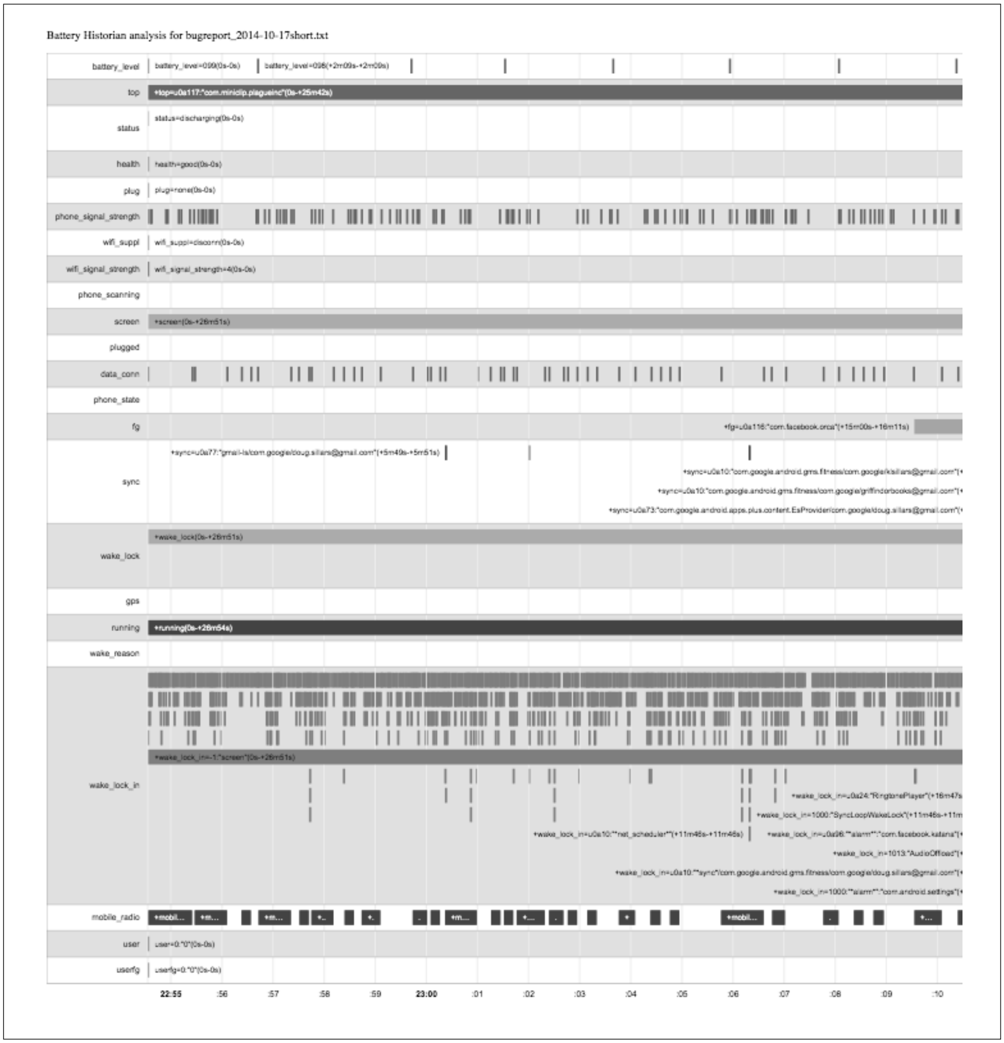


图 3-7：Battery Historian-Legacy（主视图）

正如我们看到的，原始文件很复杂难懂，里面包含了很多的数据。虽然这张表已经是简化过的，但还是需要我们仔细琢磨。放大这张表的顶部，我们可以看到图 3-8 中所示的内容。

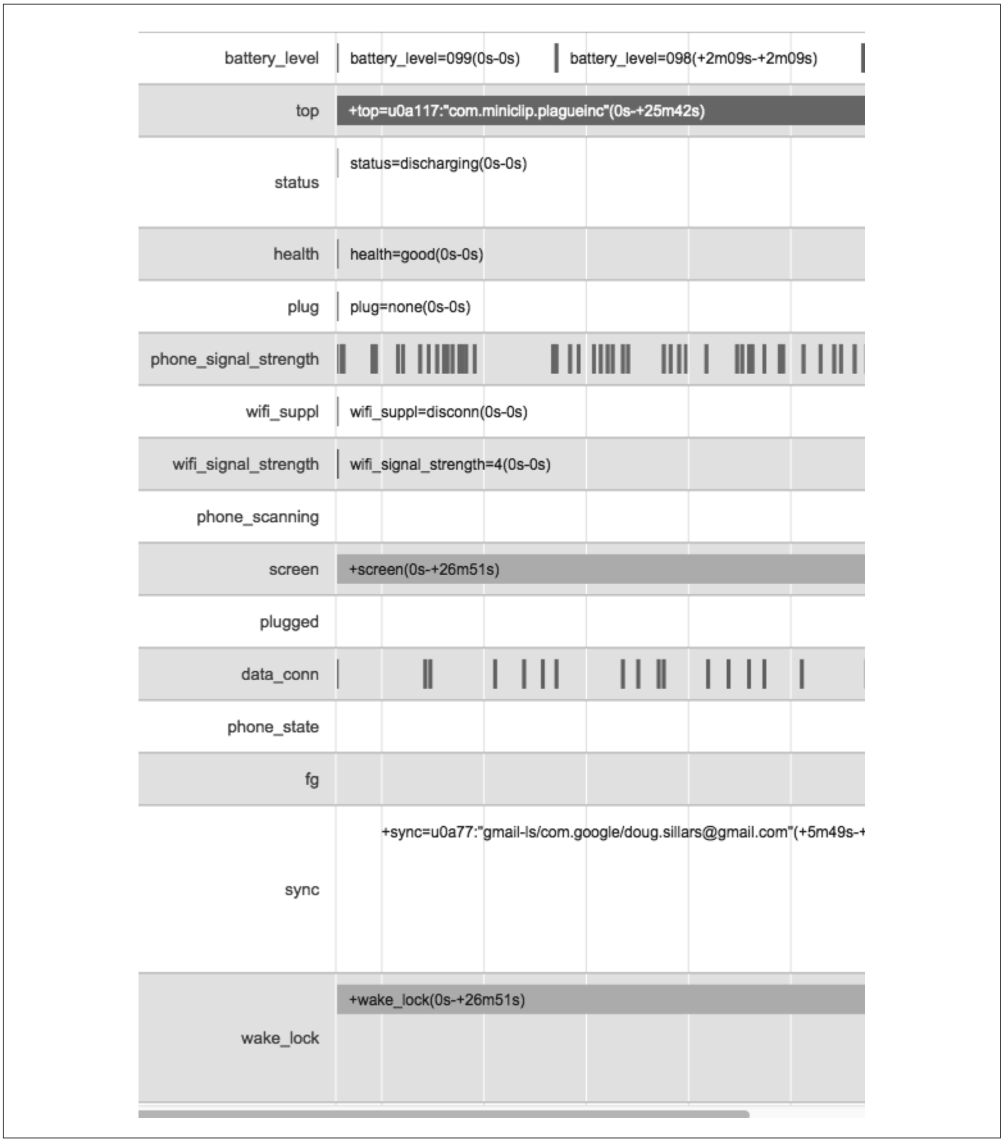


图 3-8：Battery Historian-Legacy（顶部视图）

在图 3-8 中，相邻的白色竖线表示 1 分钟的时间间隔，鼠标悬停在每一个单元的时候，会显示出这个单元的详细信息。

- battery_level（剩余电量）
  鼠标悬停在 battery_level 变化处，显示剩余电量，以及距上次 battery_level 变化的间隔（如图 3-9 所示）。

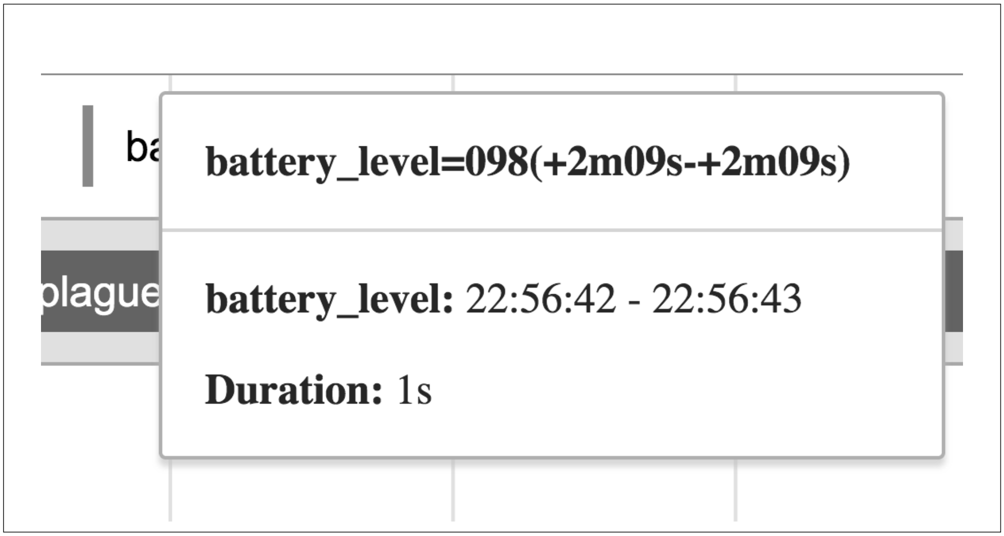


图 3-9：剩余电量变化

- `top`（上栏）
  列举了当前屏幕上显示的进程（这个例子中是游戏 Plague Inc.）。
- `Battery info`（电池信息）
  - `status`（状态）
    正在放电（与充电状态相对应）。
  - `health`（健康）
    电池健康状态，来自电池管理器 API。
  - `plug`（连接状态）
    设备是否接通电源。
- `phone_signal_strength`（无线网络信息）
  显示信号变化（有差、中、好 三种）。

鼠标悬停在信号强度的变化处，可以看到更详细的信号强度信息（见图 3-10）。

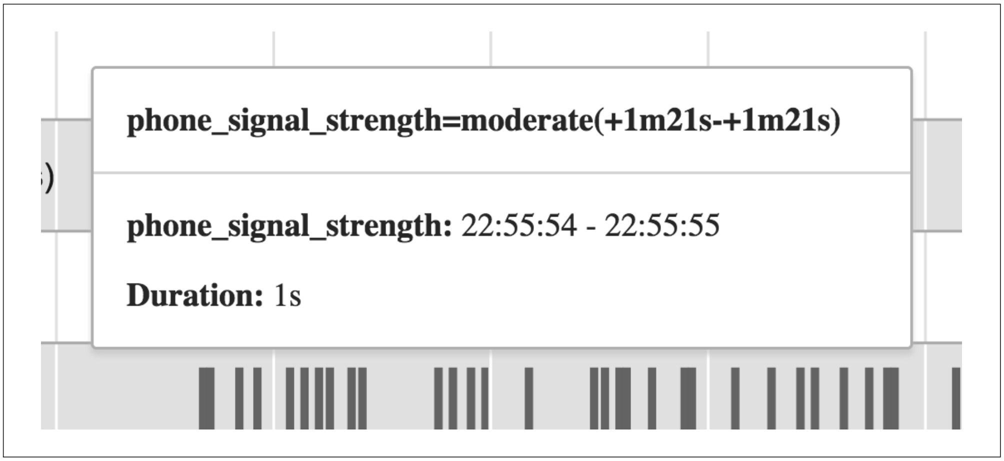


图 3-10：信号强度

- `wifi_suppl`（Wi-Fi状态）
  图 3-8 中，Wi-Fi 处于断开状态。

- `wifi_signal_strength`（Wi-Fi信号强度）
  图 3-8 中，有 Wi-Fi 信号被检测到——尽管 Wi-Fi 已经被关闭，但由于高级 Wi-Fi 设置，也会一直搜索 Wi-Fi 信号。

- `phone_scanning`（电话扫描）
  如果没有信号，手机就会扫描信号（这样会导致更多的电量消耗）。

- `screen`（屏幕）
  屏幕开启的时长。

- `plugged`（连接）
  电量来源（类似于上面的电量信息）。

- `data_conn`（数据连接）
  鼠标移动到 data_conn 部分的蓝色方块上，可以看到蜂窝数据从 HSPA 更换到了 HSPAP[3]。

将鼠标悬停在 data_conn，可以看到手机连接蜂窝无线网络的更多细节（见图 3-11）。

注 3：HSPA 和 HSPAP 是两种通信技术。——译者注



图 3-11：数据连接类型

- phone_state（手机状态）
  可以看到蜂窝网络覆盖变化，或者你是否打了一个电话。
- fg
  这里指的是前台应用。前台程序很少被销毁来释放内存。即使关闭屏幕，我们也可以看到 Facebook 客户端在前台处理收到的信息。
- sync（同步）
  处理与服务器的同步。

鼠标悬停在 sync 事件上，可以看到同步发起的原因（见图 3-12）。



图 3-12：同步发起的原因

设备上的大部分进程都见名知意。进程显示手机状态并建立电能消耗分析（包括运行的程序和执行的通话）。

在这个电池续航的表中，我们需要关注的是同步和 WakeLock 的发生频率。如果你的 App 经常唤醒设备（这样你会更容易看到你的进程名称出现在鼠标悬停的地方），你应该检查同步和唤醒设备的频率。找到消息及时性和电量消耗的平衡点是非常关键的。使用不精确的 Alarm 或 JobScheduler API 唤醒设备，可能会引发在同一时间进行多个同步或 WakeLock。正确做法是：其他 App 唤醒设备的时候，触发你的 Alarm，这样可以最小化唤醒的次数。

关闭屏幕，可以看到 WakeLock 和耗电问题的更多信息，如图 3-13 所示。

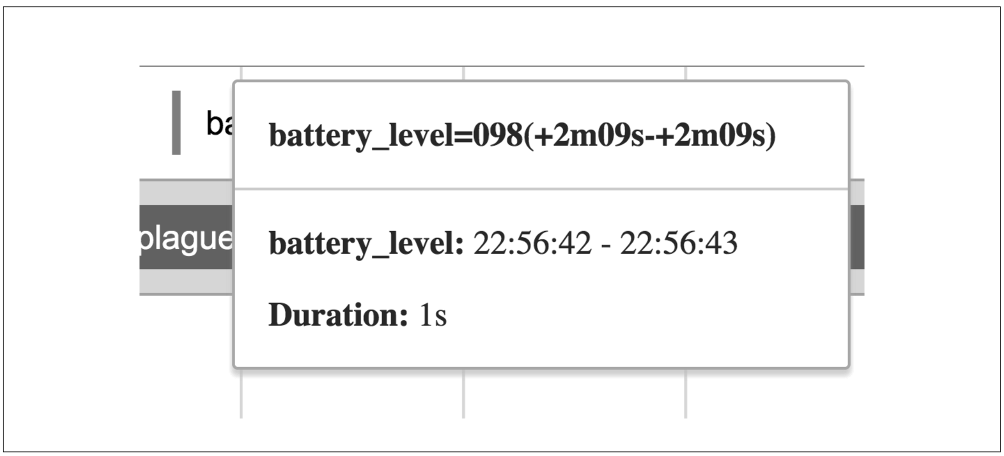


图 3-13：Battery Historian-Legacy（仰视图）

- `wake_lock`
  在游戏期间，WakeLock 一直被持有以保持屏幕常亮。

- `gps`（全球定位系统）
  当 GPS 信号打开的时候。

- `running`（运行）
  游戏时，电话一直在后台运行。

- `wake_reason`（唤醒原因）
  这一项显示了设备唤醒的原因。截图上没有显示出原因，因为设备一直处于唤醒状态。该行列出了设备上所有运行的处理器级别的进程。我们得到了以下比较常见的唤醒原因。

- qcom, smd-modem

  高通共享内存驱动，与调制解调器内存交互。

- qcom, smd-rpm

  高通共享内存驱动 - 电源管理器。

- qcom, mpm

  高通 MSM 电源休眠管理；关闭时钟，将设备置为休眠状态。

- qcom, spmi

  高通系统电源管理接口；让设备从工作状态回到休眠状态。

- wake_lock_in（唤醒锁）

  可以看到哪些进程正在运行，以及所引起的 WakeLock 或 Alarm。在这张截图中，游戏音频控制进程多次调用 WakeLock（播放不同的示例音乐引发了将近 2000 次音频 WakeLock）。同时也可以看到屏幕 WakeLock。

鼠标悬停在 WakeLock 上，就可以看到引发 WakeLock 的进程（见图 3-14）。



图 3-14：WakeLock 触发

- mobile_radio（手机无线网络）

  蜂窝无线网络的连接时间（指连接到网络，不一定传输数据），手机切换网络时会有空挡。

- user（使用者）

  可能会有多个使用者的情况。

- userfg（前台使用者）

  测试时，处于前台的使用者。

图 3-13 中可以看到唤醒手机的进程。正如我们在 3.3.7 中提到的一样，当 App 唤醒设备时，它会消耗电量。因为有一款游戏正在运行，所以截图比较枯燥。我们可以通过其他的轨迹了解一些有趣的 WakeLock 现象。

使用Battery Historian查找错误的WakeLock

如果你觉得你的 App 太过频繁地调用 WakeLock，可以使用 Battery Historian 验证。我进行了一段较长时间的 Battery Historian 测试（图 3-15 中的竖条表示 30 分钟的间隔），这期间强制关闭了一个调用很多 WakeLock 的 App[4]。

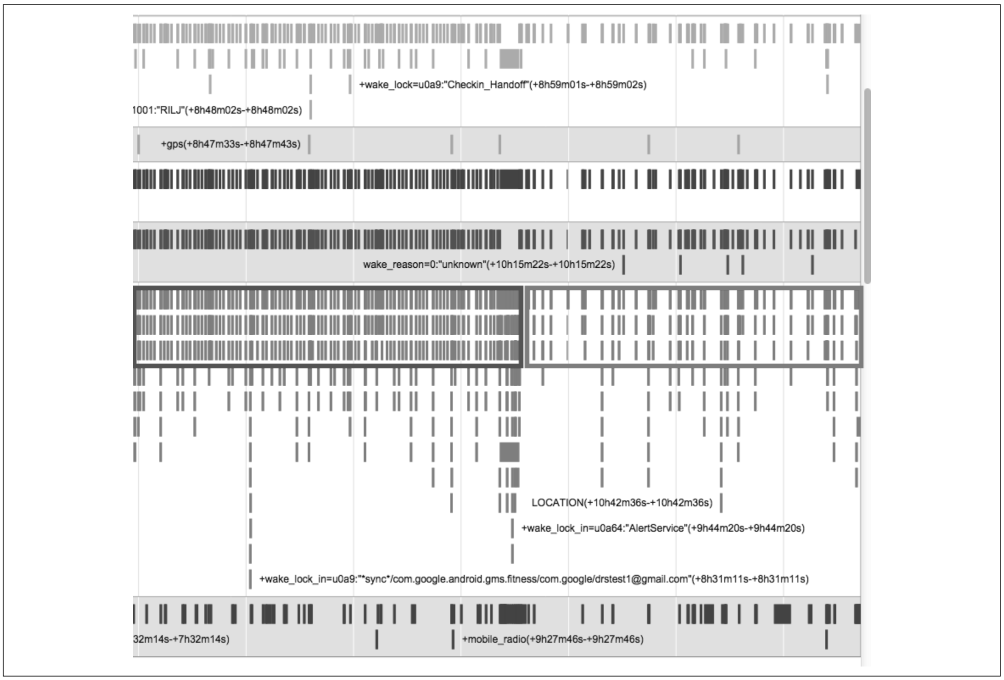


图 3-15：使用 Battery Historian 查找过于频繁的 WakeLock

诚然，使用 Battery Historian 可以很轻松地发现设备的 WakeLock 行为变化。结束问题 App 前，我们可以看到，在图中左边方框内，WakeLock 发生得非常频繁。App 每分钟至少调用三次 WakeLock，分别是：App 本身、位置传感器和加速度传感器。这些 WakeLock 一般间隔一分钟。随后在右边方框间，我关闭了这个 App，在左边方框内可以看到 WakeLock 次数和频率立即下降（WakeLock 间隔增长变为 5 分钟左右）。

下面的 Battery Historian 图表列出了测试期间的所有事件。就这个流氓 App 来说，我们可以分别在它运行和不运行的情况下运行轨迹测试。我统计出 WakeLock 数从每小时 594 次

注 4：用来作对比。——译者注

下降到 478 次。这说明了此 App 每小时引发了大约 120 次 WakeLock。一个 App 不应该频繁调用 WakeLock。要习惯通过测试确定你的 App 没有过度唤醒设备。正如 WakeLock 和 Alarm API 的说明一样，关注 WakeLock 行为至关重要，因为它们极大程度上影响了 Android 设备的耗电量。

3.5.3 Battery Historian 2.0

随着 Battery Historian 2.0（BH2）的发布，Google Android 团队完全重写了这个工具（从 Python 语言改为 Go 语言编写[5]）。3.5.2 节中，我们已经看到了新版本的所有界面，新版本更加实用，它将电量使用情况精确到了每个进程。新版本改变了使用脚本创建网页的形式，而在 9999 端口挂载了一个转换数据和提供报告的服务。快速浏览一下 BH2 上的新功能。打开一个 bugreport（故障报告）文件，就可以看到如图 3-16 所示的全新 UI。

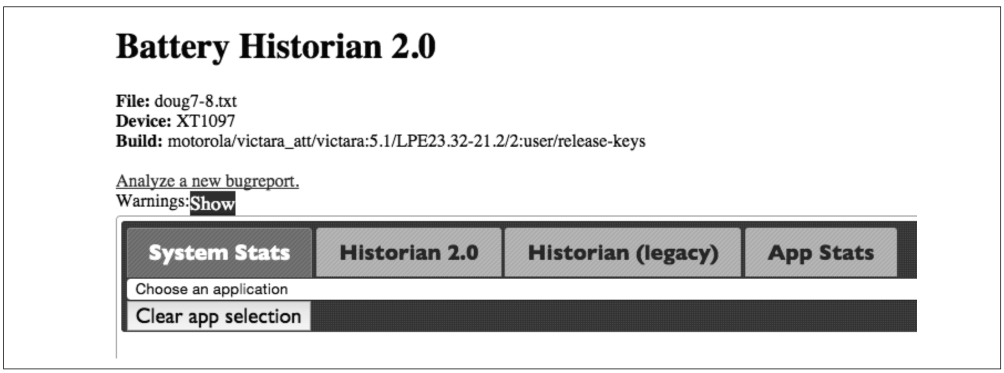


图 3-16：Battery Historian 2.0 的顶部视图

顶部显示了文件的名称、设备（摩托罗拉 X Lollipop 5.1 系统），还有四个标签：System Stats（系统统计）、Historian 2.0、Historian（legacy）和 App Stats（应用程序统计）。在看过了老版本的 Battery Historian 之后，我们来看一下三个新标签提供的信息。

System Stats 标签由多个电量消耗详情表组成。第一个表汇总统计数据（见图 3-17）：在不到 4 小时里，屏幕开启 40 分钟，设备唤醒但屏幕关闭有 24 分钟。屏幕开启每小时耗电 19%，屏幕关闭每小时耗电 4%。无线网络开启超过 2 小时，平均每小时消耗流量 2.6MB。

注 5：Go 语言是谷歌在 2009 年发布的第二款开源编程语言。——译者注

XT1097 LPE23.32-21.2

Aggregated Stats:

Metric	Value
Device	XT1097
Build	LPE23.32-21.2
Duration / Realtime	3h49m32.521s
Screen Off Discharge Rate (%/hr)	3.81 (Discharged: 12%)
Screen On Discharge Rate (%/hr)	19.31 (Discharged: 13%)
Screen On Time	40m23.648s
Screen Off Uptime	24m48.785s
Userspace Wakelock Time	8m16.784s
Kernel Overhead Time	16m32.001s
Mobile KBs/hr	2603.93
WiFi KBs/hr	0.00
Mobile Active Time	2h17m55.355s
Signal Scanning Time	0

图 3-17：Battery Historian2.0 汇总统计

仔细看的话，会发现在图 3-17 中有五行是有下划线的。每一行都对应一个表，每个表都可以按进程分类。比如，移动信号发射的开启时间和数据使用情况（见图 3-18）。

Mobile radio (active) time per app:

Ranking	Name	Uid	Mobile Active Time
0	com.levelup.touiteur		
1	GOOGLE_SERVICES		
2	com.att.connect		
3	com.google.android.googlequicksearchbox		
4	ROOT		
5	com.facebook.katana		

Mobile traffic per app:

Ranking	Name	Uid	MB Transferred Over Mobile
0	com.att.connect		
1	com.google.android.googlequicksearchbox		
2	com.android.chrome		
3	com.levelup.touiteur		
4	GOOGLE_SERVICES		
5	com.google.android.apps.inbox		

图 3-18：Battery Historian 2.0 无线网络使用统计

图 3-18 中显示的是每个应用的网络使用时长和流量。我们可以看到是哪个 App 使用无线网络时间最长。com.levelup.touiteur 程序代码（一个 Twitter 客户端 ）使用网络时间最长。

但是 com.att.connect 使用的流量最多（在摩托罗拉 X 上，每个 App 使用网络的时间和流量不是很集中，但其他设备确实存在这种情况；表中 App 的排名仍是准确的）。

进程 com.att.connect 使用大量数据并不出乎意料。测试期间，我使用这个 App 与同事进行了 30 分钟的电话会议。我没想到 Twitter 客户端使用网络的时间比我的电话会议时间还长。结合 App stats 标签的数据可以看出，相比电话会议 App，Twitter App 的连接特点如下：连接次数多，每次数据少，网络和电池使用时间长。

App	# 连接次数	KB	网络使用时间	电池使用量
com.levelup.touiteur	18	1024	23 min	5.91%
com.att.connect	6	3056	18 min	5.78%

App stats 页列出了测试期间每一个 App 使用的 WakeLock 次数（和持续时间）、服务和进程数。我的 Twitter 客户端使用了 18 次 partial WakeLock（至少它们之间有重叠），唤醒 35 个服务，使用两个进程。这些数据会帮助你挖掘你的 App 行为，同时也很适合与团队成员分享。

Historian 2.0 标签使用全新界面展示出整个能耗追踪过程中，WakeLock 和设备使用对电池寿命的影响（见图 3-19）。

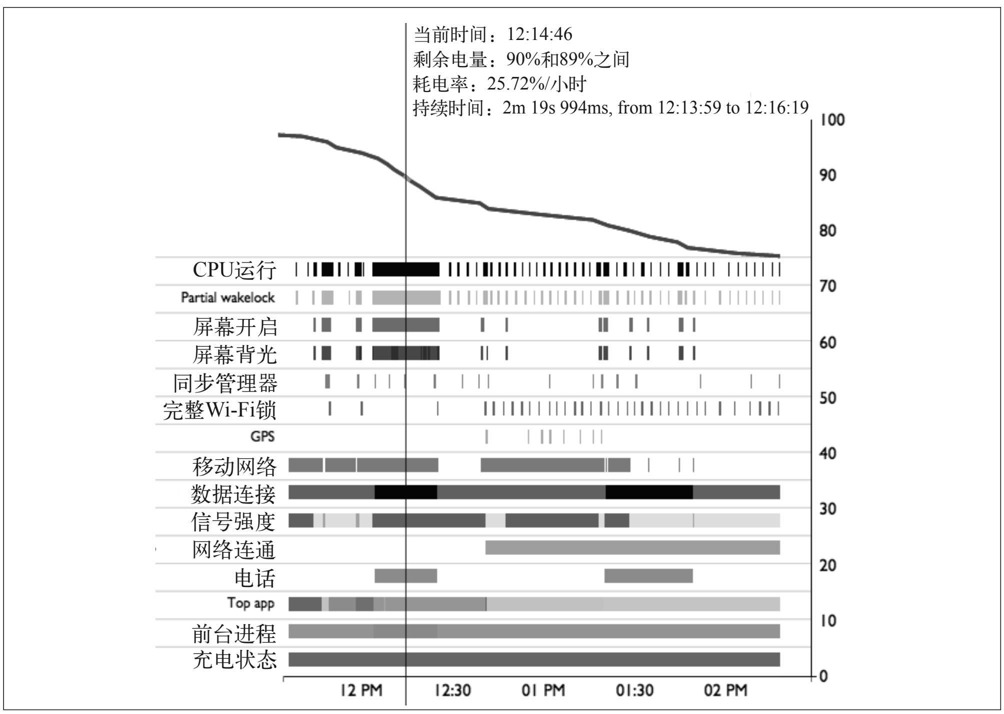


图 3-19：Battery Historian 2.0 图表

图表使用相近的方式展示 WakeLock、CPU、GPS、无线网络等，在右侧增加了一个纵轴，

数据上方的蓝色折线显示了电量消耗。能耗的每一个百分点都可以选中，选中后会显示这期间的统计信息。在图 3-20 中，纵坐标表示电量消耗。在我进行电话会议时，电量急速下降（两分钟消耗 1% 或者每小时消耗约 25%）。

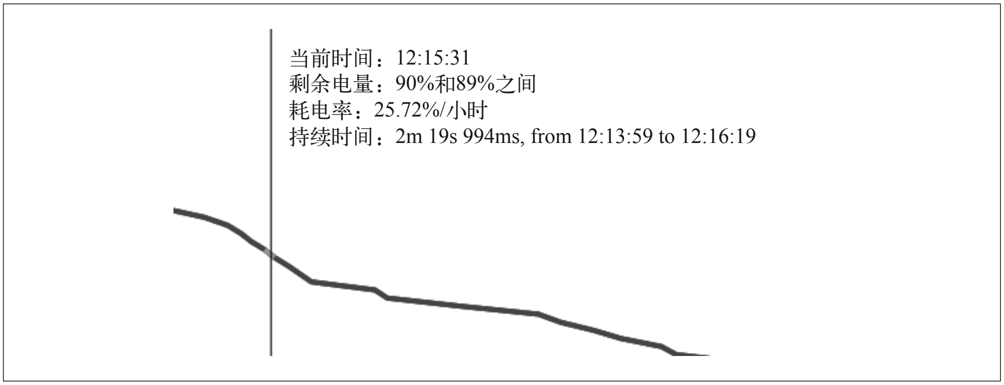


图 3-20：Battery Historian 2.0 的耗电详情

最初的 Battery Historian 工具提供很多设备层面的统计，帮助我们确定每个独立 App 的行为。2.0 版本添加的新功能使得挖掘单进程的数据更简单。现在你可以很轻易地分析出你的 App 中引起电量消耗的功能，然后解决问题。

3.6 JobScheduler

在 Lollipop 版本，Android 增加了一个新的 API ——JobScheduler（作业调度器）。JobScheduler 可以替代 WakeLock 和 Alarm 运行 App 的任务。可以将它看作"互相协作的 WakeLock/Alarm"API。每个 App 中的 WakeLock 和 Alarm 都是相互独立的，但是 JobScheduler 将设备的唤醒抽离至操作系统层面。因为 Alarm 和 WakeLock 受沙箱限制，所以无法与安装在设备上的其他应用互相协调。如果 5 个 App 每 30 分钟唤醒设备一次，则它们的唤醒几乎不可能同步，最终设备每小时会被唤醒 10 次。但由于 JobScheduler 是在系统层级，系统可以更有效地执行所有的调度工作。每小时唤醒设备的次数也会减少。

除了调度设备唤醒，JobScheduler 还允许设定获取数据的时间间隔，比如把唤醒时间限制在 8 分钟之后 10 分钟之内。这给操作系统留出一定的调整范围，让系统可以更好地协调唤醒以达到省电的目的。也可以说，App 不仅可以提前拿到想要的数据，还可以节省电量（对 App 来说是双赢）。想象一个天气 App 每 10 分钟连接一次网络（每小时 6 次）。在使用无线网络的情况下，为了节约电量，提前一些更新数据会对 App 产生影响吗？许多情况下，这只意味着数据更新比 App 需要的更快，但却节省了宝贵的电量。在如下的代码片段中（源自我修改的 JobScheduler，http://github.com/dougsillars/HighPerformanceAndroidApps），我设置最小的连接时间间隔为 7 分钟，又强制最晚 10 分

钟（到 deadline 的时候）：

```
JobInfo.Builder builder = new JobInfo.Builder(kJobId++, mServiceComponent);
      //kJobId允许同时运行多个JobScheduler
      <snip>
            String delay = mDelayEditText.getText().toString();
              //从UI阅读延迟时间
              if (delay != null && !TextUtils.isEmpty(delay)) {
                  builder.setMinimumLatency(Long.valueOf(delay) * 1000);
              }
              String deadline = mDeadlineEditText.getText().toString();
                              //从UI阅读deadline时间
              if (deadline != null && !TextUtils.isEmpty(deadline)) {
                  builder.setOverrideDeadline(Long.valueOf(deadline) * 1000);
            }
    boolean requiresUnmetered = mWiFiConnectivityRadioButton.isChecked();
    boolean requiresAnyConnectivity = mAnyConnectivityRadioButton.isChecked();
    if (requiresUnmetered) {
        builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED);
    } else if (requiresAnyConnectivity) {
        builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY);
    }

    builder.setRequiresDeviceIdle(mRequiresIdleCheckbox.isChecked());
                      //仅在设备空闲时会通过复选框强制运行JS
    builder.setRequiresCharging(mRequiresChargingCheckBox.isChecked());
                      //只在充电时通过复选框强制运行JS
    mTestService.scheduleJob(builder.build());
```


从代码中可以看到 JobScheduler 还有些其他有用的功能（通过复选框控制）：

- 运行一个周期服务，维持网络连接
- 仅在不限流量的网络下运行某项工作（一般指 Wi-Fi）
- 仅在设备空闲时运行（API 中对于空闲状态说得不是很明确，只是说设备“有时候”会处于空闲状态）
- 当设备接通电源时运行
- 连接频率衰减；延长后续连接的间隔时间

假设你的 App 每 15 分钟唤醒设备检测服务器更新。一小时内就会唤醒 4 次（每天 96 次）。如果用户还安装了一个天气 App，每 6 分钟更新一次呢（10 次 / 小时，240 次 / 天）？你的 App 和安装的天气 App 同时连接的几率极低——因为两个 App 时间不同步，两个 App 也没有交互。如果两个 App 都使用 JobScheduler API，操作系统将协调 App 达到省电的目的。在 2014 年 Google 开发者大会上展示了 Volta 项目，Google 估算在每个 App 都使用这个 API 的情况下，可以节约 15%~20% 的电量。

我对 Android SDK JobScheduler 的范例做了扩展，使得 JobScheduler 与它的一些额外功能协调工作。这个任务是从服务器上下载图片，如图 3-21 所示，我设定 App 每 60 秒下载一

张图片（API 将在 60 秒间隔内建立一个连接，但不一定是 60 秒），14 分钟内这个 App 将响应 14 次。

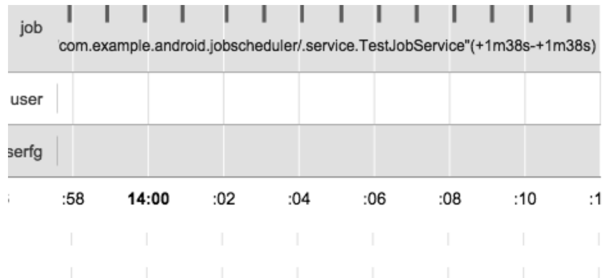


图 3-21：使用 JobScheduler 的连接周期（设置为 60 秒）

图 3-22 是一个类似的测试，设置下载间隔是 150 秒（时间轴放大倍数相同），这样连接 6 次耗时超过 14 分钟。

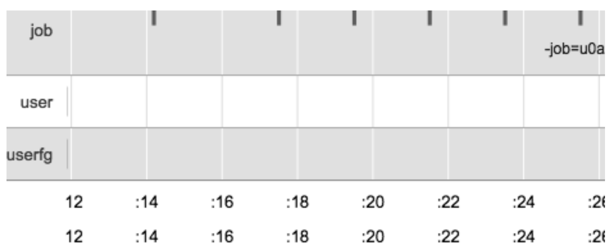


图 3-22：使用 JobScheduler 的连接周期（设置为 150 秒）

使用传统的 WakeLock，我们假设 App 同时运行且 App 之间互不可见，设备就会发起 20 次连接，因为这些连接很难重叠。如果使用 JobScheduler 同时运行这两项工作，系统将同步这些周期性的连接来节约电量。只用 9 次连接就达到了 20 次的效果。



图 3-23：同步 60 秒和 150 秒的连接

JobScheduler 另一个很炫酷的特点就是执行重复工作，并且可以指定执行的周期是线性还是指数性衰减。如果 App 不处在前台，你可能不需要持续这种频繁的更新。因此，你可以让它们降低更新频率。当 App 重新打开的时候，用户仍然可以看到最新的数据，后台数据使用量却减少了。

JobScheduler 有两种延时工作的衰减方式：线性（慢衰减）和指数性（快衰减）。线性衰减获取当前的 deadline，增加衰减时间（fallbacktime）乘以（失败数 −1）。如图 3-24 所示，deadline 是 20 秒，衰减延时也是 20 秒，所以接下来的每次 ping 间隔都会增加 20 秒（调度到开始分别是 20、40、60…），指数回退增加回退时间乘以 $2^{(失败数-1)}$，延时以 2 的指数次增长（调度到开始分别是 20、60、180、325，645）。现在，你的 App 可以更新数据了，并且这种方式使用的数据更少，更节省电量。

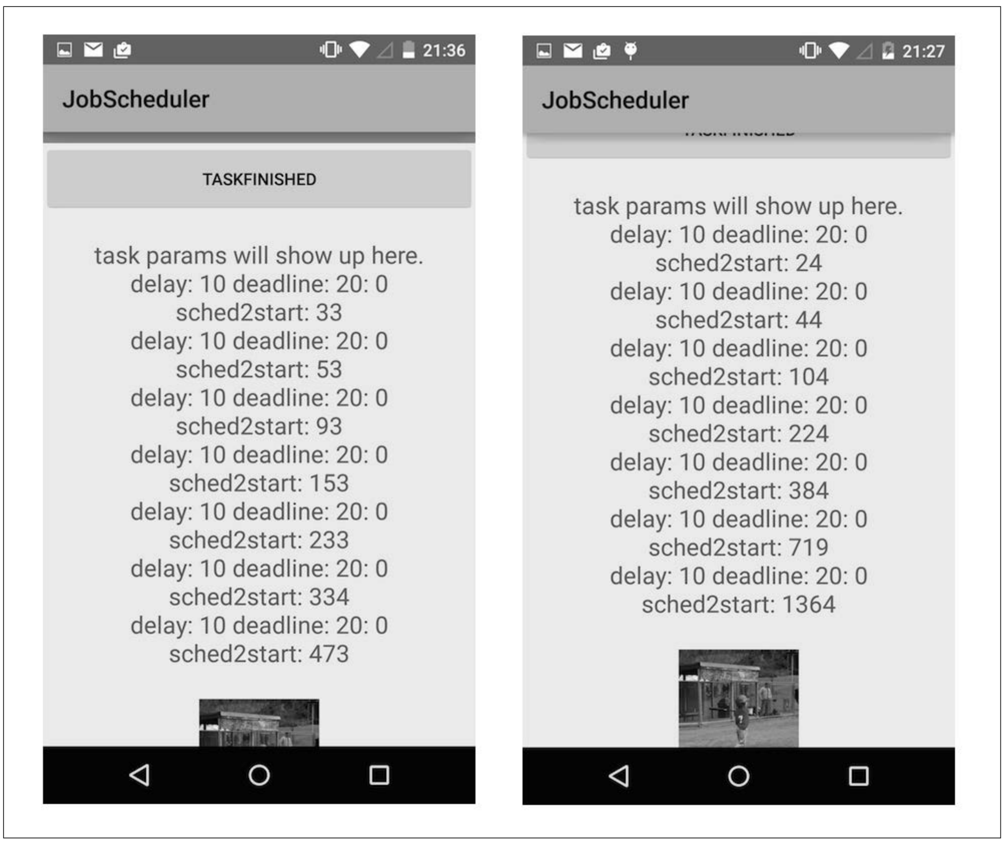


图 3-24：使用 JobScheduler 的 App 截屏，线性（左）衰减和指数（右）性衰减

很明显，让操作系统来调度你的 App 中非重要的工作，对延长电池寿命是有巨大帮助的。当我写这篇文章的时候（2015 年 4 月），Lollipop 只占有 Android 设备的 5.5%，但随着数量的增长，更多的用户将从使用 JobScheduler API 中受益。

3.7 小结

电池续航能力是衡量 App 性能的重要指标，电量消耗多的 App 通常表现为经常唤醒设备，或不让设备进入休眠状态。我们一起了解了 Android 如何计算每个 App 的电能消耗（基于硬件测量），以及用户如何发现问题 App（帮助你避免 App 出现在问题列表中而被用户卸掉）。我们还看了 Android Lollipop 中的新版 Battery Historian 工具，它可以给开发人员提供更多更详细的 Android App 电量消耗数据，查找过度唤醒设备的 App。此外，我们还研究了 JobScheduler 如何通过操作系统统一调度多个 App，以达到减少后台唤醒次数的目的。

第 4 章

屏幕和UI性能

App 的 UI 可能会受到设计师、开发人员、易用性研究者以及测试人员的影响——似乎每个人都喜欢对 App 界面给出建议或反馈。UI 将用户和 App 连接起来，是 App 的门面，因此需要精心地设计 UI。然而，不管 App 的 UI 是简单还是复杂，能高效地运行才是最重要的。

作为开发人员，你的任务是和 UI/UX 设计团队合作，在每一个 Android 设备上完美地展现他们的设计理念。前文已经简要地讨论了 Android 多屏幕尺寸适配的问题，那么 UI 性能呢？设计团队设计的（你实现的）UI 如何在 App 上展现？页面加载是否快速？ UI 响应是否迅速、流畅？在这一章，我们将讨论如何优化 UI，使其能够快速渲染并流畅运行，也会介绍一些分析屏幕和 UI 性能的工具。

4.1 UI性能基准

所有的性能优化首先都要确定目标，UI 性能优化也是如此。“我的 App 需要加载得更快”这样的说法固然是好的，但是最终用户的期待是什么，这些期望能落实的又有多少呢？一般情况下，我们可以参考人际交往的心理学指标。研究显示，0~100 毫秒的延迟会让用户感知到瞬时的卡顿；100~300 毫秒的延迟会让用户感觉迟缓；300~1000 毫秒的延迟让用户感觉“手机卡死了”；1000 毫秒以上的延迟会让用户想去干别的事情。

由于这是基本的人类心理感知，上述的标准也可以认为是衡量页面 / 视图 /App 加载时间的一个不错的指标。Ilya Grigorik 曾经做过一个很棒的演讲（https://www.youtube.com/watch?v=Il4swGfTOSM），介绍如何搭建秒开的手机网站。如果网页可以在 1 秒内完成加

载，你就战胜了人类的感知，可以凭借网站丰富的内容来吸引用户了。另外有研究表明，50% 以上的用户已经开始放弃使用那些加载时间在 3~4 秒的网页了。对于 App 也是如此，App 加载得越快，用户体验也就越好。在本章中，我们将关注 UI 的加载时间。至于那些可能必须在后台运行的任务，比如从网络下载文件等，我们将会在后面的章节中介绍其优化方法（或者是防止这些任务阻碍渲染的方法）。

卡顿

内容的快速加载很重要，渲染的流畅性也很重要。Android 团队把滞缓、不流畅的动画定义为卡顿，一般是由丢帧引起的。大部分 Android 设备一秒刷新屏幕 60 次（也有例外，比如早期的 Android 设备的刷新频率是 50fps，甚至更低）。由于屏幕每 16 毫秒刷新一次（1s/60fps=16ms/f），所以保证每帧的渲染时间少于 16 毫秒是非常重要的。如果有一帧跳过了，用户就会感知到动画的跳跃，这样的体验是非常不好的。为了保证动画的流畅度，我们将研究如何在 16 毫秒内完成整个屏幕的渲染。在这一章，我们将分析一些常见的问题并说明如何保证 UI 不出现卡顿现象。

4.2 Android上的UI和渲染性能改进

用户对早期 Android 版本的主要吐槽点之一就是 UI，尤其是触摸交互和动画的卡顿。因此，随着 Android 越来越成熟，开发者投入了大量的时间和精力使 UI 尽可能更快、更流畅。接下来我们看看 Android 各个版本都对 UI 性能做了哪些改进。

- 在使用 Gingerbread 或更早的 Android 系统版本的设备上，屏幕绘制是完全在软件上完成的（没有 GPU 的需求）。然而，随着 Android 设备的屏幕越来越大，像素精度越来越高，为了能及时渲染屏幕，对软件的要求也越来越高。
- Android 的 Honeycomb 版本新增了平板电脑版，进一步扩大了屏幕的尺寸。考虑到这一点，这个版本增加了 GPU 芯片，App 可以选择完全使用 GPU 硬件加速来运行渲染的程序。
- 对于运行在 Ice Cream Sandwich 或者更高版本系统上的 App，GPU 硬件加速是默认打开的，App 会将大部分渲染工作交给特定的硬件，这显著地提高了渲染的速度。
- Jelly Bean 4.1 和 4.2 版本上，为了让 App 运行起来更加流畅，“黄油计划”为避免卡顿和抖动做了进一步的改进。通过改善 VSYNC 的时序（更好地调度帧的创建），增加额外的帧缓冲，Jelly Bean 设备改善了卡顿丢帧的情况。Android 团队在做这些改进的同时，也提供了一系列的工具来测量屏幕绘制、VSYNC 缓冲和卡顿的情况，并对开发者开放了这些工具。

回顾一下所有的改变、引入的工具，以及它们对于普通 Android 开发者的意义。正如你所想的一样，这些改进的目标是：

- 降低屏幕绘制的延迟

- 创建流畅、稳定的帧率以避免卡顿

当 Android 团队致力于改进屏幕渲染和 UI 性能时，他们需要一些工具来量化对操作系统所做的改进。值得表扬的是，他们将这些工具集成到了 Android SDK 中，因此普通的开发者也能够使用这些工具来测试 App 的渲染性能问题。下面我们用一些小例子来解释这些工具是如何工作的。

让我们开始动手吧！

4.3 创建视图

这里假设你知道如何在 Android Studio 上创建 XML 布局并利用 Android Studio（Eclipse）上的各种工具查看这些视图。如图 4-1 所示，你会看到一个简单的 App，里面包含一系列嵌套的视图。在创建视图的时候，可以在屏幕的右上角查看组件树。视图嵌套的层级越多，视图树就越复杂，渲染的时间也就越长。

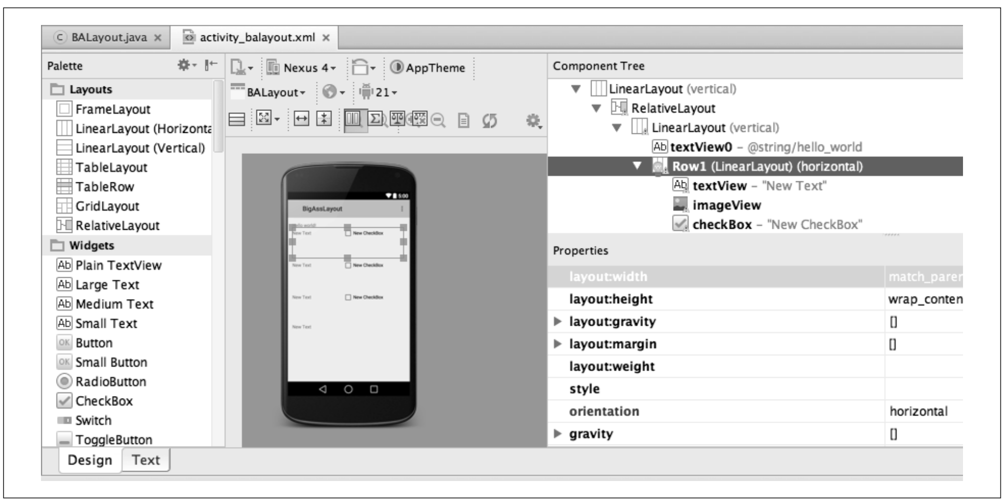


图 4-1：App 布局的设计视图

对于 App 上的每一个视图，显示在屏幕上需要经过三个步骤：测量、布局和渲染。想象一下 App 是如何绘制一个 XML 布局的：从顶部节点开始测量，然后根据布局树逐个渲染：测量每个视图在屏幕上的尺寸（在图 4-1 上就是先绘制 LinearLayout，然后是 RelativeLayout，LinearLayout，之后是分支 textView0 和 LinearLayout Row1。Row1 还有三个子视图）。每一个子视图都会向父视图提供尺寸，好让父视图确定摆放的位置。如果父视图发现它的尺寸测量有问题（或者是子视图的尺寸不对），它可以强制每一个子视图（或者是子视图的子视图）重新测量以解决问题（有可能增加一两倍的测量时间）。这就是设计一个少嵌套的视图树的价值所在。视图树的层级越多，嵌套测量的次数越多，测量计

算的时间就会越长（特别是需要重新测量的时候）。我们将通过几个例子来说明，在浏览视图时，重新测量是怎样增加渲染时间的。

重复测量视图

重复测量视图并不一定是因为错误。RelativeLayout 就需要经常对它的子视图测量两次，以确保所有子视图被放置在了正确的位置。如果 LinearLayout 的子视图设置了 layout weight 属性，那么 LinearLayou 也需要测量两次以确定子视图的确切尺寸。如果是嵌套的 LinearLayout 或者是 RelativeLayout，测量的次数会呈指数增长（两层嵌套会进行 4 次测量，3 层嵌套会进行 8 次测量，等等）。图 4-9 就是一个非常好的重复测量的例子。

一旦视图测量完成，每一个视图都将会对自己的子视图进行布局，子视图布局完成后，回到父视图，最后回到根视图。布局完成后，每个视图会被绘制在屏幕上。注意，是所有的视图都被绘制，而不仅是用户能看到的那些。我们将会在 4.4.1 节中讨论屏幕过度绘制的问题。App 的视图越多，就需要越长的时间测量、布局和绘制。为了减少这些任务花费的时间，尽可能地保持视图层级的扁平化并删除所有不必要渲染的视图是非常重要的。要通过减少布局的层级来加快屏幕的绘制，我们要做的还有很多。理想情况下，全部的测量、布局和绘制的时间最好在 16 毫秒以内，这样才能保证屏幕运行的流畅性。

如图 4-1 所示，虽然可以在 XML 布局文件里查看布局的节点视图，但是很难找到多余的视图。为了找到这些多余的视图（还有那些增加屏幕渲染时间的视图），我们可以使用 Android Studio Monitor（Android Studio 的一部分，但是可以作为一个独立的应用程序运行）中的 Hierarchy Viewer 工具分析 Android App 里的视图，从而解决这些问题。

Hierarchy Viewer

Hierarchy Viewer（层次结构查看器）能够很便捷地以可视化方式查看各种视图嵌套关系，可用于研究 XML 视图结构。Hierarchy Viewer 可以在 Android Studio Monitor 中使用，并且需要一个运行 Android App 的设备。可在 2.4 节的“root 机器 / 工程师 / 开发者构建”部分获取更详细的信息。Google Romain Guy（http://github.com/romainguy/ViewServer）中的一个类可以帮助开发者测试调试版本的 App。接下来的几个小节中，所有的布局和截图来自于 Android 4.1.2 版本的 Samsung Note II。通过在老设备（较慢的处理器）上进行屏幕渲染测试，可以确定是否解决了此类设备的渲染问题。这样可以保证 App 在所有 Android 设备上出色渲染。

如图 4-2 所示，打开 Hierarchy View 之后，你可以看到这里有很多窗口：左侧“Windows”标签下列出了所有与电脑连接的 Android 设备，以及所有正在运行的进程。粗体展示的是活动的进程。第二个标签列出了当前正在编译的进程的详情（后面将会介绍）。中间部

分是可缩放的视图树形图。点击其中的一个视图（这个例子中是最左边的视图），能够看到它在设备上的显示以及附加的数据。右边是两个视图：Tree Overview 和 Layout View，Tree Overview 展示了完整的视图层次结构，里面的一个方块显示了中间窗口在整个树形结构中所处的位置。Layout View 中深红色高亮的区域表示所选中视图的绘制部分（浅红色展示的是父视图的）。

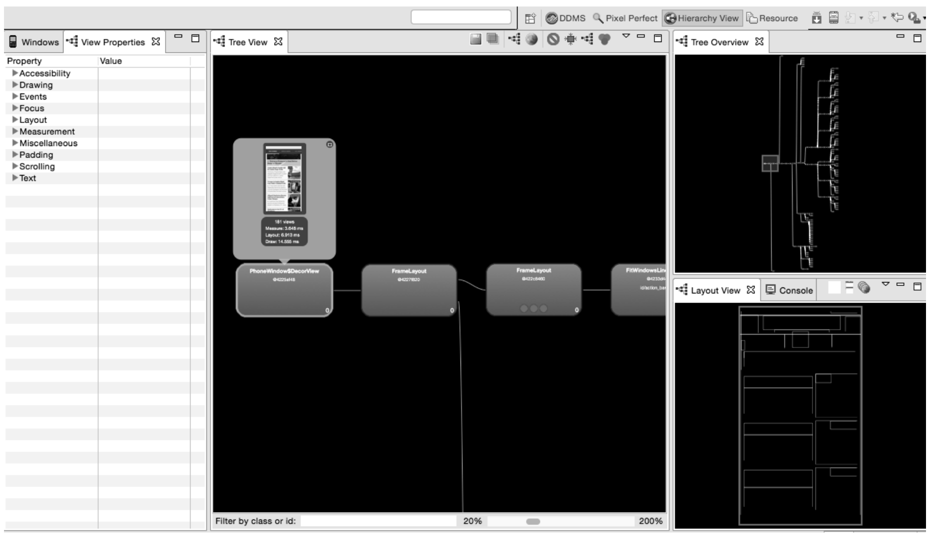


图 4-2：使用树形视图分析一款新闻 App 时，Hierarchy View 工具的界面概览（另见彩插）

在中间的窗口中，你可以点击任何一个视图来查看该视图在 Android 设备屏幕上的展示。点击 Tree View 下面工具栏里红、绿、紫三色的 Venn 图图标，会有弹窗显示子视图的个数以及视图的测量、布局和绘制的总用时。这个总用时是该视图及其所有子视图进行渲染所花费的时间总和（在图 4-3 中，我选中了顶部的视图来获取整个视图的创建时间）。

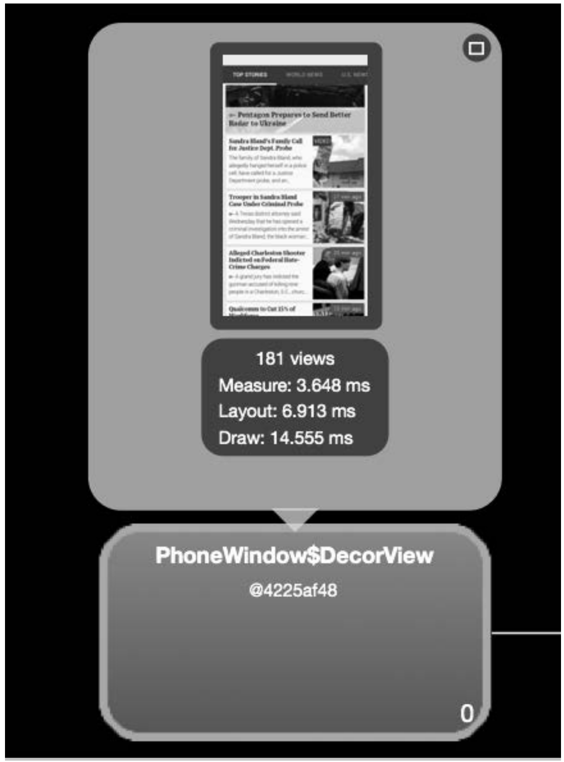


图 4-3：视图的渲染时间

图中文章列表最上方的视图包含了 181 个子视图，视图测量花费了 3.6 毫秒，布局用了 7 毫秒，而绘制用了 14.5 毫秒（总共大约 25 毫秒）。要减少这些视图的渲染时间，我们可以通过查看这个 App 的 Tree Overview 来了解这些视图是如何组成一个整体的。从 Tree Overview 中可以看到，屏幕上有很多个视图，渲染树的结构相对扁平。渲染树越扁平越好，因为视图 XML 文件的深度越深，渲染所需的时间就越长。然而，图中的结构虽然是扁平的，可还是需要 26 毫秒来进行绘制，这说明扁平的结构也有可能会卡顿，也需要考虑如何优化（正常情况下 16 毫秒才算流畅）。

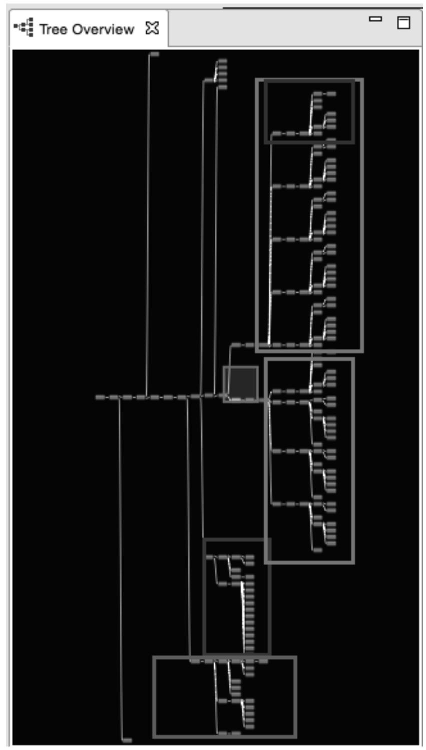


图 4-4：Tree Overview（另见彩插）

图 4-4 展示了一个新闻 App 的文章列表页的 Tree Overview，可以看到，它主要包含三个区域：头部（位于视图底部的蓝色框中），文章列表（两个橙色框里面是两个不同的文章标签）。单篇文章的视图是用红色方框高亮显示的，内部标题视图的结构重复出现了 9 次（顶部的橙色框中有 5 次，第二个橙色框中有 4 次）。最后，我们可以看到底部的侧边导航栏的视图（在绿色框中）。头部使用了 22 个视图，两个文章列表分别使用了 67 个和 44 个视图（每个头部标题使用了 13 个视图），而导航栏使用了 20 个视图。这样还剩下 18 个视图没有被计算在内。其中包含一个滑动动画以及一些用来完成动画的临时视图。很显然，视图的数量会积少成多，要实现无卡顿的用户体验，必须要保证视图的渲染尽可能地高效。

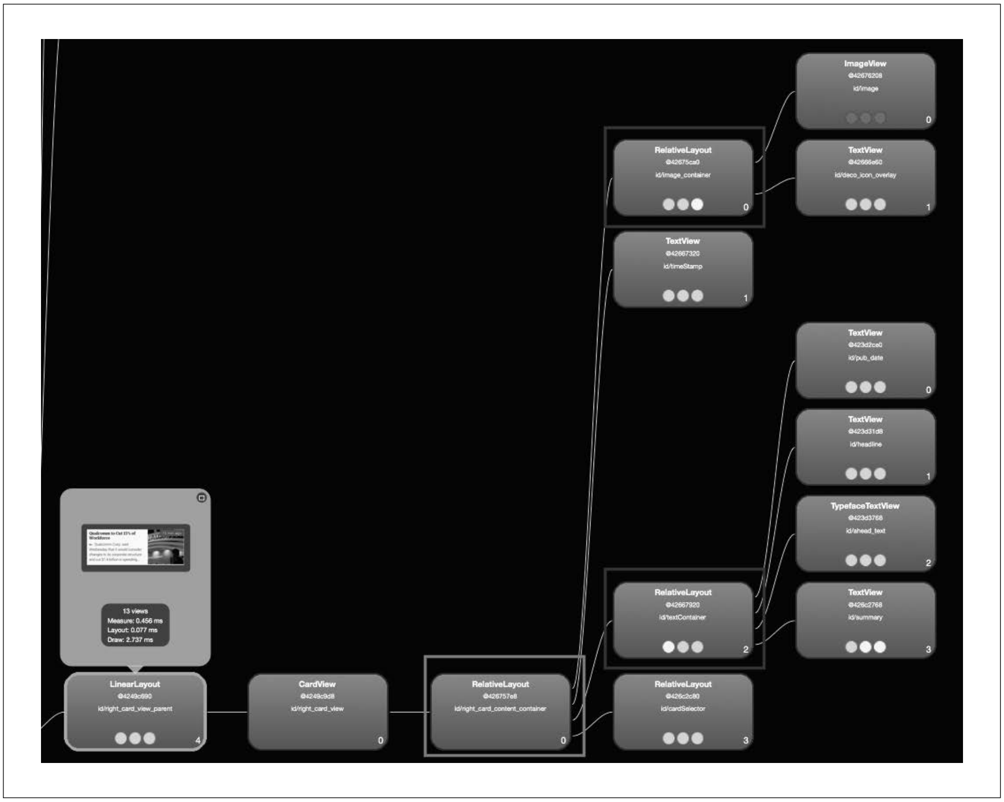


图 4-5：分析一个视图树（另见彩插）

仔细看标题部分，可以看到一个标题由 13 个视图组成。每个标题结构有 5 层（视为垂直列），需要 0.456 毫秒来测量，0.077 毫秒来布局，2.737 毫秒来绘制。层次结构中的第五层是第四层中的两个 RelativeLayout（蓝色高亮显示）的子视图，而这两个 RelativeLayout 又是第三层的另一个 RelativeLayout（绿色高亮显示）的子视图。如果我们把第四层的两个 RelativeLayout 的子视图都移到它们的父视图中（位于第三层），那么整个第五层就不用渲染了。而且，正如我在 4.3 节中所解释的那样，每个 RelativeLayout 会被测量两次，因此嵌套的 RelativeLayout 会导致测量时间迅速增长。

现在，你也许注意到了每个视图中的红色、黄色和绿色圆点。它们表示该视图在那一层树形结构的视图里测量、布局和绘制的相对速度（顺序是从左往右）。绿色代表最快的前 50%，黄色代表最慢的前 50%，而红色表示那一层所有视图里面最慢的视图。很显然，红色标记的视图是一个很好的优化点。

再来看文章标题的树形结构，绘制最慢的视图是右上角的 ImageView。顺着这个 ImageView 一直找到文章父视图，我们可以看到它是两个 RelativeLayout（会增加测量时间）的子视图，然后是三个没有子视图的视图（在底部）。这三个视图也可以优化合并成

一个视图，这样能减少两层的渲染。

下面我们来看看另一个新闻 App 是如何减少每个标题使用的视图数量的。图 4-6 展示了一个同图 4-5 类似的层次结构图。

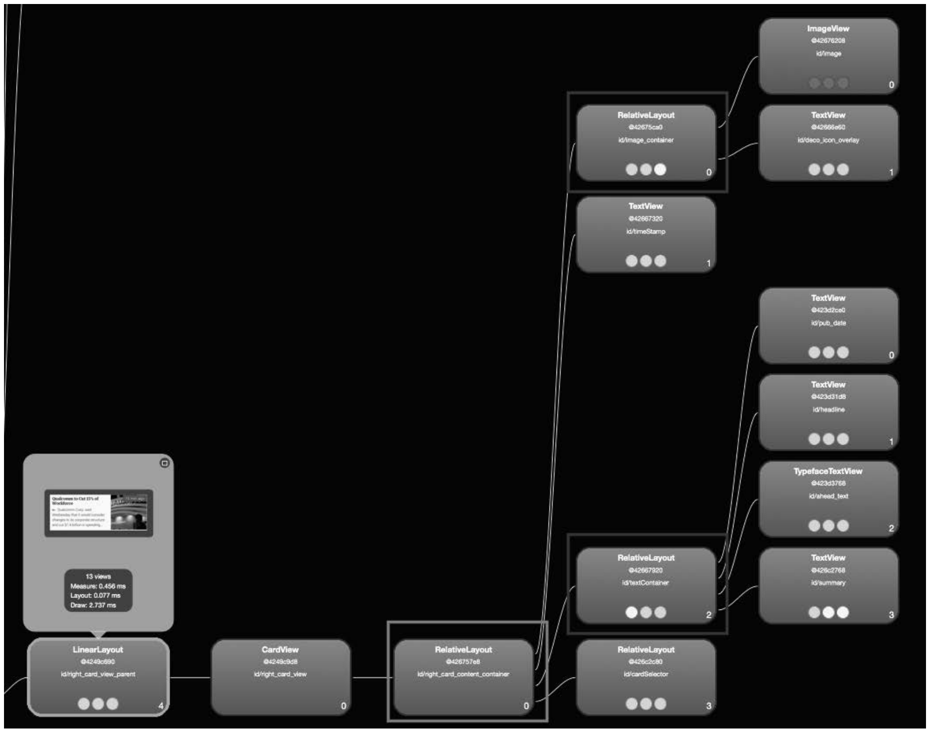


图 4-6：新闻文章的原始视图树

事实上，图 4-6 中的标题视图也有 RelativeLayout 的问题，这导致它测量需要 1.275 毫秒，布局需要 0.066 毫秒，而绘制需要 3.24 毫秒（一个标题共需要 4.6 毫秒的渲染时间）。在这些数据的基础上，开发人员又做了一些调整，加入一个更大的图片和一些分享按钮，但是整个层次结构更扁平了（如图 4-7 所示）。



图 4-7：更新后的视图树

现在的主标题（三层的层次结构）一共只用了 4.2 毫秒进行渲染，展示的内容更大了，但渲染时间却减少了 400 毫秒。

为了更好地研究性能方面的问题，我将使用示例 App“Is it a goat?”中的一些例子。这个简单的 App 是几张山羊图片（旁边带有选中标识）的列表。该 App 构建了几种不同的布局，包括性能差的和性能好的。通过分析视图以及它们的优化过程，我们能够清楚地了解如何优化 App 的渲染性能。我们分几步对这个 App 进行优化，通过在设置中切换视图在 Hierarchy View 中查看每一次的变化。每选择一个布局类型，视图将会由一个更优（或者更劣）的 XML 视图刷新结构。我们把“Slow XML”布局作为未优化的开始节点。先快速地浏览这个 App 未优化版本的 Hierarchy View，如图 4-8 所示。

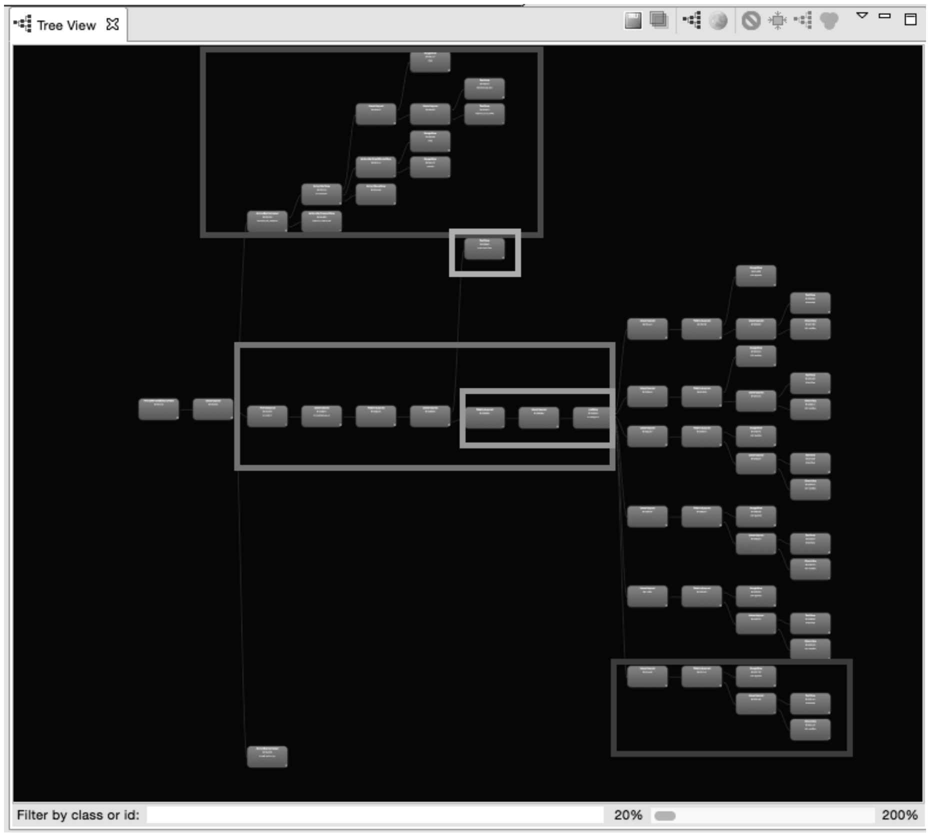


图 4-8：未优化的“Is it a goat？”App 的 Hierarchy View（另见彩插）

这个简单的 App 中有 59 个视图。和图 4-4 里的新闻 App 不同，这个 App 的视图树的水平深度更深。一个视图的子视图越多，渲染就会越费时。减小视图树的深度，这款 App 每一帧的渲染就会更快。

蓝色框内是 Android Action Bar 的视图，橙色框是屏幕顶部的文本框，紫色框里展示的是山羊的详细信息（紫色框里有 6 个一样的视图）。红色框一行显示了 7 个视图，这除了增加 App 的深度外，并没有其他作用。仔细观察绿色框中三个连续的视图，可以发现存在重复测量的问题（见图 4-9）。



图 4-9：Hierarchy View 中的重复测量

当设备开始测量视图的时候，先从右边的子视图开始，然后到左边的父视图。右边的 ListView 包含 6 行数据，一共 37 个视图，花了 0.012 毫秒来测量。把这个 ListView 加到中间的 LinearLayout 之后，变成 38 个视图。有趣的是，测量时间由于循环的重复测量而爆表了。测量时间连升三个数量级，达到了 18.109 毫秒。LinearLayout 左边的 RelativeLayout 使得测量时间加倍，变成了 33.739 毫秒。加上多余父视图（位于图 4-8 的红色框中）的测量时间，总的测量时间超过了 68 毫秒。但只要简单地移除中间这个 LinearLayout，整个视图树的测量时间瞬间降低到了 1 毫秒以下！（你能通过在示例 App 中将“Remove Overdraw”布局同“Remove LL+OD”布局进行比较，从而看到这个变化。它们唯一的不同就是移除了这个 LinearLayout）我们能够通过采用“Optimized Layout”设置移除更多的层。最终结果如图 4-10 所示，只有 3 层了。

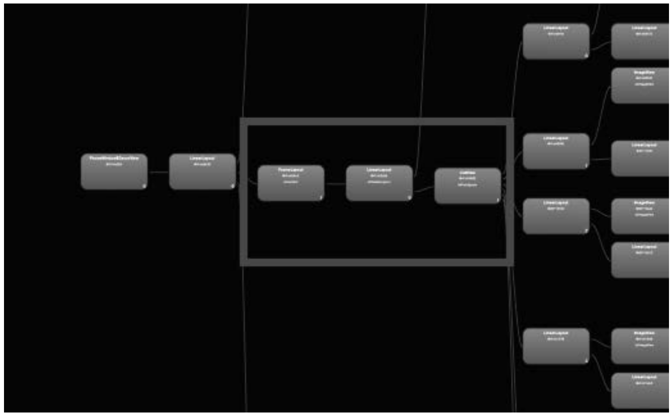


图 4-10：移除层次结构深度

通过查看山羊数据可以发现一个进一步优化视图深度的方法。每一行山羊信息有 6 个视图，屏幕中显示了 6 行的山羊信息（这样的一行数据在图 4-8 中的紫色框底部高亮显示）。使用 Hierarchy View 工具查看一行山羊信息的视图是如何在 App 中构建的（见图 4-11），我们能够看到最左边的两个视图（一个 LinearLayout 和一个 RelativeLayout）仅仅只是增加了视图的深度（当前显示“Slow XMl”视图）。原始的 LinearLayout 直接加入到了 RelativeLayout 中，但并没有展示其他什么内容。

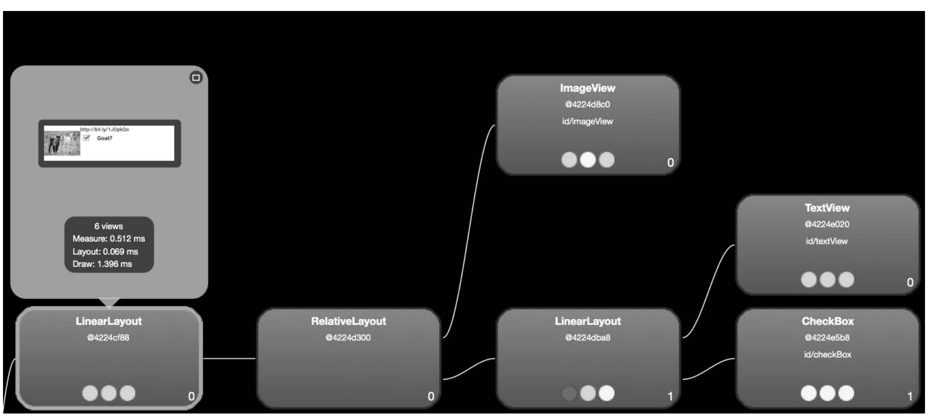


图 4-11：一行山羊信息的未优化的 Hierarchy View

因为 RelativeLayout 会重复测量 2 次（我们正在尽量减少测量时间），所以我首先尝试移除 RelativeLayout（App 中设置“Optimized Layout”，如图 4-12 所示）。这样做后，深度从 4 变成了 3，渲染变快了。

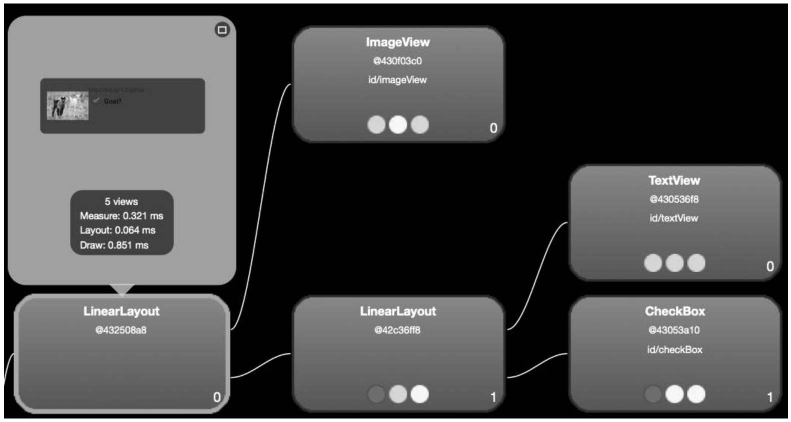


图 4-12：移除 RelativeLayout 后的视图

但效果还不理想。通过移除 LinearLayout，同时调整 RelativeLayout 来显示整行的信息（如图 4-13 所示），视图深度降低到了 2。布局的渲染又加快了 0.1 毫秒。这告诉我们优化布局的方法不止一种，不妨多尝试几种不同的选择（参考表 4-1）。

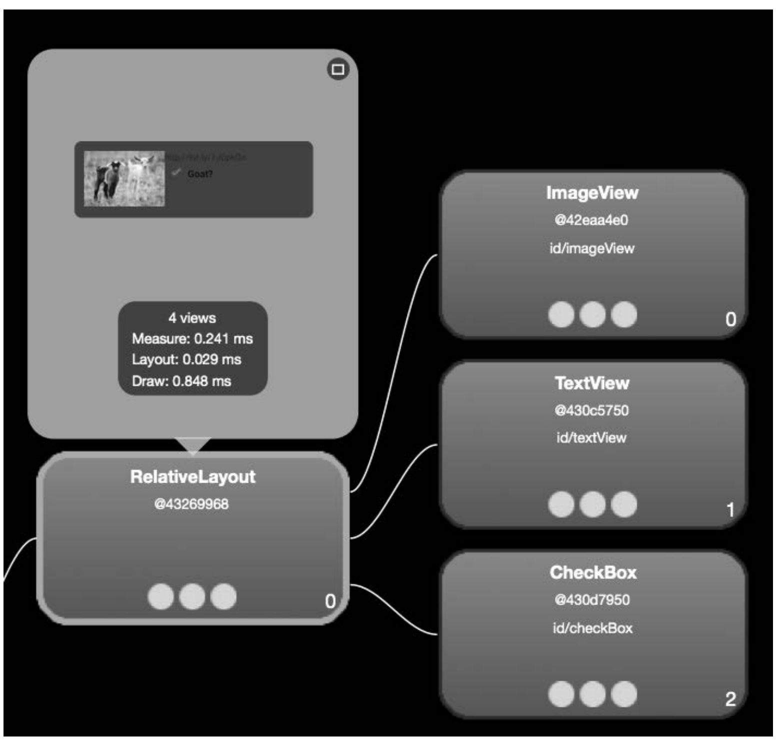


图 4-13：合并到 RelativeLayout 后的视图

表4-1：视图树的优化改进

版本	视图数	视图深度	测量	布局	绘制	总用时
Unoptimized	6	4	0.570	0.068	1.477	2.115 ms
Remove RelativeLayout	5	3	0.321	0.064	0.851	1.236 ms
Remove LinearLayouts	4	2	0.241	0.029	0.848	1.118 ms

通过将每一行信息的渲染时间减少大约 1 毫秒，整体视图的渲染时间将会减少 6 毫秒（假设屏幕上有 6 行数据）。如果 App 存在卡顿或者测试显示 App 渲染时间已接近 16 毫秒了（卡顿边界），那么节省 6 毫秒意味着 App 将远离卡顿。

视图重用

优秀的面向对象的程序员会尽可能地重用视图，而不是反复重新创建。在“Is it a goat?”App 中，每一行展示的布局都是重用的。如果 XML 文件里最外层的视图只是用来承载子视图的，那么它只是增加了 App 视图结构的深度。这种情形下，你可以移除这个视图，用 `<merge>` 标签来代替。这样就能删除 App 层次结构中多余的层次。

作为练习，在 GitHub 上下载“Is it a goat?”App，并且在 Hierarchy View 工具中观察视图的渲染时间。你能通过改变设置中的单选按钮来修改使用的视图 XML 文件，然后通过 Hierarchy View 工具展现 App 的深度变化以及这些变化对 App 渲染速度的影响。

Hierarchy Viewer（不止是树形结构图）

Hierarchy Viewer 还有一些附加功能能够帮助我们更好地理解重绘。从左到右看一下树形结构窗口的选项，可以发现下面这些功能。

- 把任意视图的树形结构图保存为 PNG 图片（图标是一个格式化的软盘）。
- 导出为 Photoshop 格式（将在 4.4.1 节中进行说明）。
- 视图重载（第二个紫色的图标树）。
- 在另一个窗口中打开大的视图（球状图标）；还可以设置背景色，用来发现是否存在过度绘制。
- 使视图失效（有条红线穿过的按钮）。
- 让视图重新布局。
- 让视图在 LogCat 中输出绘制命令（紫色树形按钮的第三个用处），这样可以查看绘制到底触发了哪些 Open GL 命令。这个功能对 Open GL 的专家做深度优化比较有用。

很显然，Hierarchy Viewer 是优化 App 视图必不可少的分析工具——很可能会帮助 Android App 减少几十毫秒的渲染时间。

4.4 资源缩减

将 App 的视图结构变扁平，减少视图的数量之后，我们还可以尝试减少每个视图里使用的资源数量。2014 年，Instagram 将标题栏使用的资源数量从 29 个减少到了 8 个（http://instagram-engineering.tumblr.com/post/97740520316/betterandroid）。启动时间缩短了 10%~20%（因设备而异）。他们通过资源着色缩减资源，也就是加载时仅加载一个资源，然后在运行时通过 ColorFilter 对资源进行着色。比如，传入要使用的 drawable，然后通过下面的方法来对它进行着色：

```
public Drawable colorDrawable(Resources res,
  @DrawableRes int drawableResId, @ColorRes int colorResId) {
    Drawable drawable = res.getDrawable(drawableResId);
    int color = res.getColor(colorResId);
    drawable.setColorFilter(color, PorterDuff.Mode.SRC_IN);
    return drawable;
}
```

这样一个文件就能用来表示几种不同的对象状态了（加星或者不加星，在线或者离线等）。

4.4.1 屏幕的过度绘制

每过几年，就会有传闻说某个博物馆在用 X 光扫描一幅价值连城的名画之后，发现画作的作者其实重用了旧的画布，在名画的底下还藏着另一幅没有被发现的画作。有时候，博物馆还能用先进的图像技术还原画布上的原作。Android 的视图绘制使用了类似的方式。当 Android 系统绘制屏幕的时候，首先绘制父视图，然后是子视图，再是子视图的子视图等。子视图位于其父视图的上方，这样整个子视图就都被绘制在了屏幕上，就好像画家和他的画布那样，这些父视图都被其子视图覆盖住了。

文艺复兴时期，很多伟大的画家都要等画干了之后才能重用画布。在我们高科技的屏幕上面，屏幕重绘的速度要快好几个数量级，但是多次的屏幕绘制还是会增加延迟，并且很有可能导致布局卡顿。重绘屏幕的行为被称为过度绘制，下面我们会讨论如何检测过度绘制。

过度绘制还带来了另一个问题，任何时候只要视图失效了（当视图内容有更新的时候），视图的每一个像素都需要重绘。因为 Android 系统不知道哪个视图是可见的，所以只能重绘这些像素相关的所有视图。类比上面画家的例子，画家只能把老画一幅幅还原出来，再一层层画到画布上，最后再画上最新的画。如果 App 有多层或者有很多的视图被绘制在那些像素点上，那么你必须重绘每一个视图。一不小心，屏幕的绘制（重绘）就有可能造成性能问题。

4.4.2 检测过度绘制

Android 提供了一些很好的工具来检测过度绘制。在 Jelly Bean 4.2 里，开发者选项菜单里增

设了 Debug GPU Overdraw 工具的选项。如果你用的是 Jelly Bean 4.3 或者 KitKat 设备，在屏幕的左下角会有一个计数用来展示屏幕过度绘制的程度。我认为这个工具对快速检测过度绘制问题还是十分有效的。只不过有时候这个工具会多提示 6~7 次（发生的概率还不小）。

图 4-14 中的截图还是来自上面的“Is it a goat?”App。在左下方可以看到过度绘制的计数。屏幕中可以看到 3 个过度绘制的计数，其中开发者能控制的是主窗口的计数，显示在左下方。左边没优化过的 App，过度绘制的次数是 8.43，优化后的 App 的过度绘制次数降到了 1.38。我们还可以看到，导航栏过度绘制的次数是 1.2（菜单按钮是 2.4），也就是说文字和图标的过度绘制贡献了额外的 20%。尽管过度绘制计数可以在不过多影响用户体验的前提下，快速、便捷地比较不同 App 的过度绘制情况，但没办法定位过度绘制问题具体是在哪产生的。

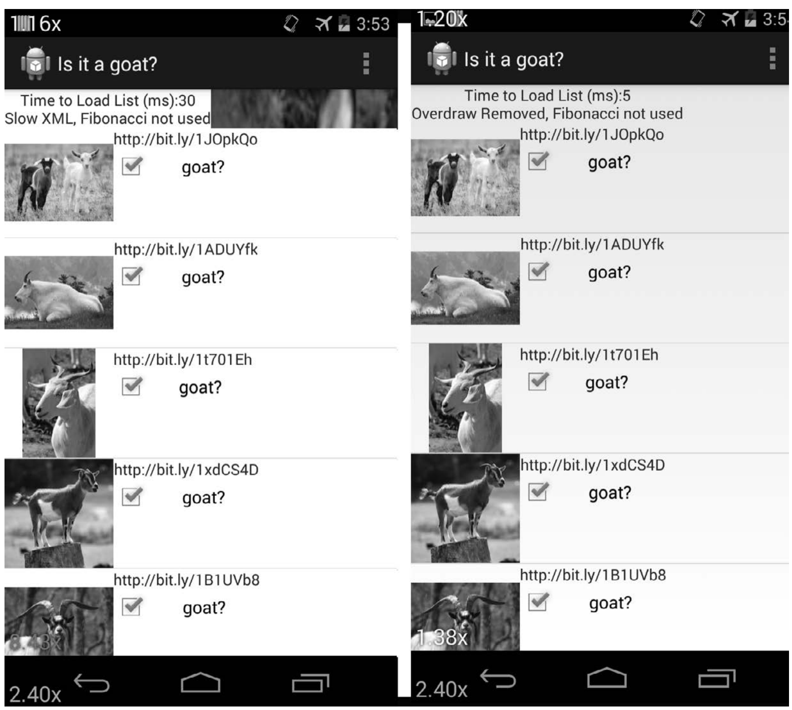


图 4-14：同一 App 未优化（左）和优化后（右）视图的过度绘制计数

另一种查看过度绘制的方式是在 Debug GPU Overdraw 菜单里选择“Show Overdraw areas”选项。选择之后，会在 App 的不同区域覆盖不同的颜色来表示过度绘制的次数。比较屏幕上这些不同的颜色，可以快速、方便地定位过度绘制问题。

- 白色

  没有过度绘制

- 蓝色

  1 次过度绘制（屏幕绘制了 2 次）

- 绿色

  2 次过度绘制（屏幕绘制了 2 次）

- 浅红色

  3 次过度绘制

- 深红色

  4 次或者更多次过度绘制

在图 4-15 中，可以看到“Is it a goat?”App 优化前后过度绘制区域渲染效果。App 的菜单栏优化前后都没有被着色（没有过度绘制），但 Android 图标和菜单按钮图标都是绿色的（2 次过度绘制）。山羊图片列表在优化之前是深红色的（表示有 4 次或以上的过度绘制）。优化 App 之后，只有复选框和图片区域是蓝色（1 次过度绘制）的了，说明至少移除了 3 层的绘制！文本和空白区域都没有过度绘制了。

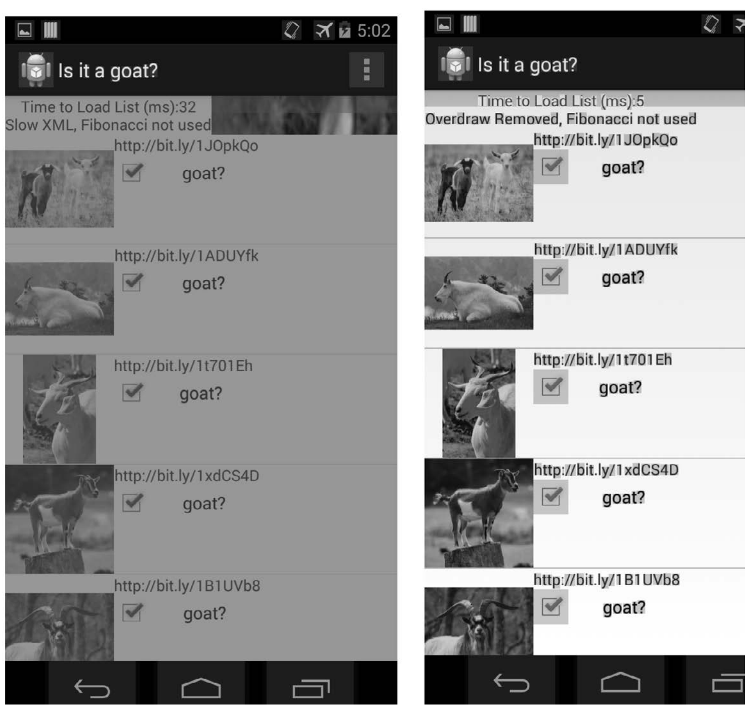


图 4-15：优化前（左）和优化后（右）的过度绘制颜色

通过减少视图的数量（或者至少替换过度绘制的视图），App 的渲染速度会加快。通过

比较 Hierarchy Viewer 中示例 App 的过度绘制版本和优化后版本（Slow XML 与 Remove Overdraw）的父视图的绘制时间，可以发现优化后的绘制时间减少了一半，由 13.5 毫秒降到了 6.8 毫秒。

4.4.3 Hierarchy Viewer中的过度绘制

另一种查看 App 当中过度绘制的方式是把 Hierarchy Viewer 中的 view hierarchy 保存为 Photoshop 文档（Tree View 里的第二个选项）。如果你没有安装 Photoshop，还有几个其他的免费软件也可以打开这个文档（下面的截图来自 Mac 版的 GIMP）。打开这些视图，可以清楚地看到不同层次的过度绘制情况。对于大部分的线上 App，在一个白色背景上叠放另一个白色背景很常见。这听起来还好，但其实有一次绘制是多余的，可以避免的。在“Is it a goat?”App 中，为了让这种情况更显而易见，所有过度绘制区域都使用一张驴子的照片替换了之前的白色背景图片。看不到之前的驴子是因为被上面的白色背景图挡住了。移除掉之后就可以看到下面的驴子了，这样我们就可以快速地定位哪里出现了过度绘制，并移除过度绘制。用 GIMP 打开文档之后，App 里所有可见的视图都有一个靠近该层的小眼睛图标。在图 4-16 中，我从“Is it a goat?”App 的最上面开始把视图一个个隐藏起来了（展现了一只大驴子）。在右边的布局视图中，可以看到一些其他的全屏布局（都显示了相同的驴子图片）。



图 4-16：形象的剥落视图

图 4-17 展示了另一种逐步隐藏视图的办法。从左上角的全屏图片开始，移除两行山羊图片还有布局（可以得到第一行的第二张图片），能够看到每一行山羊数据的下面都有一张被拉伸了的驴子图片。在这些驴子图片的下面是一张白色的背景图（从第一行第三张图片可以看出）。再移除这张白色背景可以看到一张大驴子图片（如左下角的第四张图片所示）。再往下则是最底部的一张白色的全屏背景图了（右下角第五张图片）。

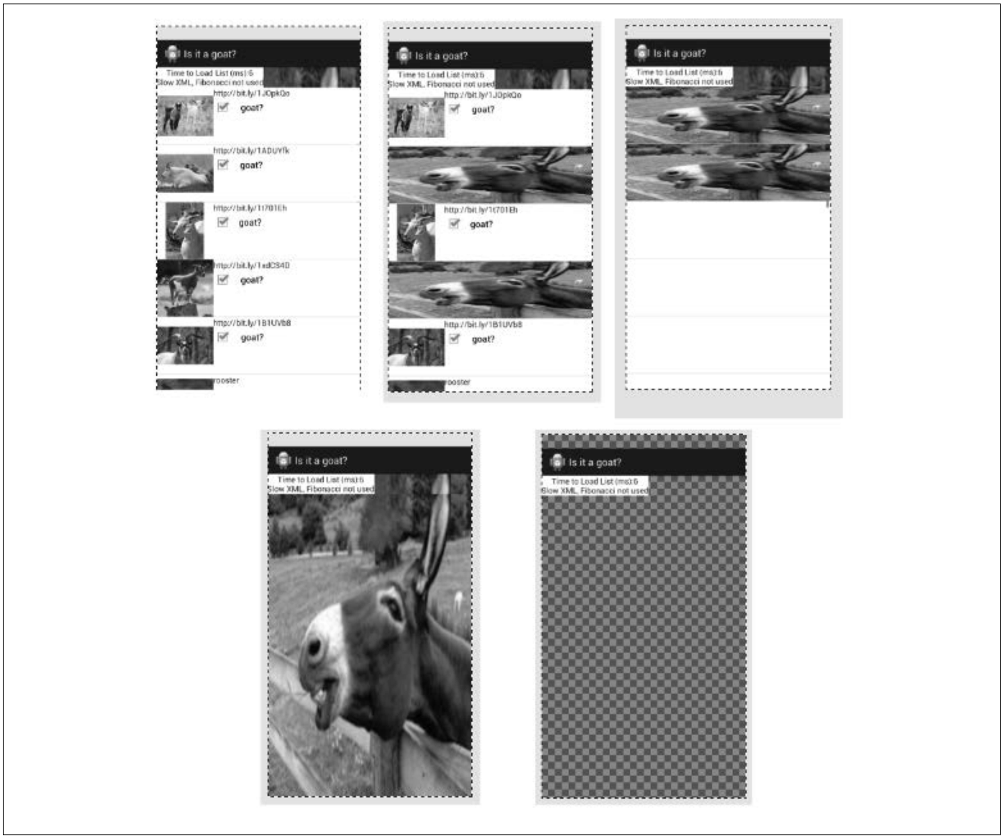


图 4-17：可视化地查看层次

4.4.4　过度绘制和KitKat（Overdraw Avoidance）

在 KitKat 或者更新版本的设备里，过度绘制的影响被大幅度地削减了。这项技术被称为 Overdraw Avoidance，系统可以自动地移除简单的过度绘制（比如一个视图被其他视图完全覆盖住了），也就是说在 KitKat 或者更新的设备里，“Is it a goat?” App 中那张全屏的驴子背景图就不会被绘制了。显然，有过度绘制问题的 App 的绘制速度会提高。但开发者还是要尽可能地避免过度绘制（为了更好地编码，也为了那些使用 Jelly Bean 及更旧设备的用户）。

Overdraw Avoidance 和相关开发者工具

当用上述提到的过度绘制检测工具时，KitKat 的 Overdraw Avoidance 功能会被禁止，因此你看到的还是原始布局，而不是系统优化后的布局。

4.5 分析卡顿（测量GPU的渲染性能）

在优化视图的层次结构和过度绘制之后，App 还是有丢帧或者滑动不流畅的现象：App 依然存在卡顿。可能在高端机器上感觉不到卡顿，但在低端机上还是能感觉到的。为了能获取更全面的卡顿检测信息，Android 在 Jelly Bean 及更新的系统里增设了一个 Profile GPU Rendering 的开发者选项。它能够测出绘制每一帧在屏幕上用了多少时间。测量数据可以保存到日志文件（`adb shell dumpsys gfxinfo`）中，或者在设备的屏幕上实时显示（只支持 Android 4.2 以上的设备）。

为了快速分析这到底是怎么回事，我比较喜欢在屏幕上直接展示 GPU 的渲染数据，这样能更全面直观地看到正在发生的事情（但日志里面的数据更适合离线画图或者作报告）。老样子，我们最好在不同的设备上都试一下。图 4-18 展示的是“Is it a goat?”App 在 Lollipop 的 Nexus 6（左边）和 KitKat 的 Moto G（右边）上运行时的 GPU 渲染概况。屏幕底部会出现颜色条，GPU 概况图表中最重要的特征就是底部的水平绿条。它表示设备渲染一帧需要 16 毫秒，每一帧是一个水平条。如果很多帧都超过了这条绿线，那就表示设备出现卡顿问题了。从下面的图片中可以看到，当页面滑动到底部，并且设备出现一个弹跳动画的时候，在 Nexus6 上运行的 App 会发生卡顿。但最终用户体验不算太糟。每一次的屏幕绘制（竖线）被分成四种颜色，用来收集一些额外的测量数据：draw（蓝色）、prepare（紫色）、process（红色）、execute（黄色）。在 KitKat 和更早的版本里，prepare 的数据还没有独立出来，包含在其他项里面（因此在 KitKat GPU 概况截图里只能看到 3 种颜色）。

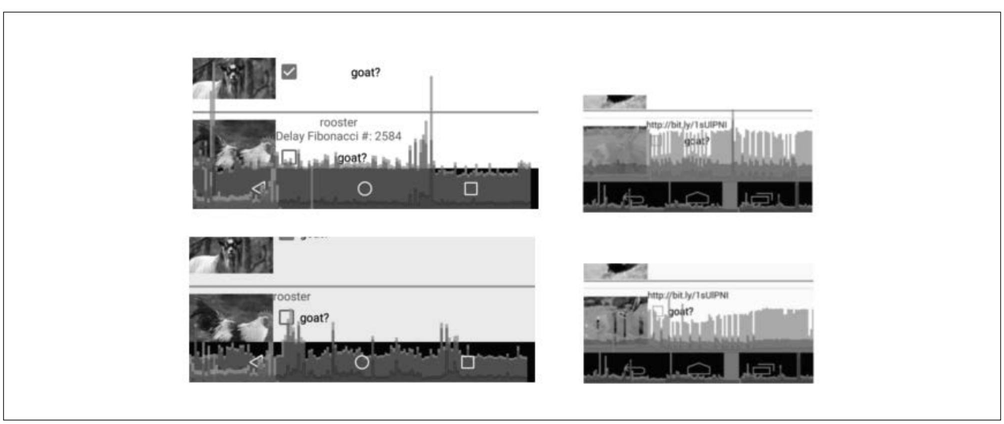


图 4-18：“Is it a goat?”App 的 GPU 概况。Lollipop 上（左）和 KitKat 上（右），未优化的（上）和优化后的（下）

让我们回到设备测试的主题，对比一下 Nexus 6 和 Moto G 的 GPU 数据。图 4-18 中，未优化过的“Is it a goat?” App 的图表精确地表明，Moto G 上的绘制时间是 Nexus 6 的两倍（尺度相同，通过比较两图中绿线的高度得到结果）。这一点可以通过数据采集（`adb shell dumpsys gfxinfo`）来进一步说明。图 4-19 所示的另一个例子当中，优化过的视图在 Moto G 上的绘制时间差不多是在 Nexus 6 上的两倍。对于两台设备来说，draw、prepare 和 process 这几步都花了差不多的时间（总共少于 4 毫秒）。差别出现在 execute 阶段（紫色），Moto G 比 Nexus 6 多用了 4 毫秒左右。这说明 GPU 渲染测试最好是在低端机器上做，这样比较容易发现视图渲染问题。

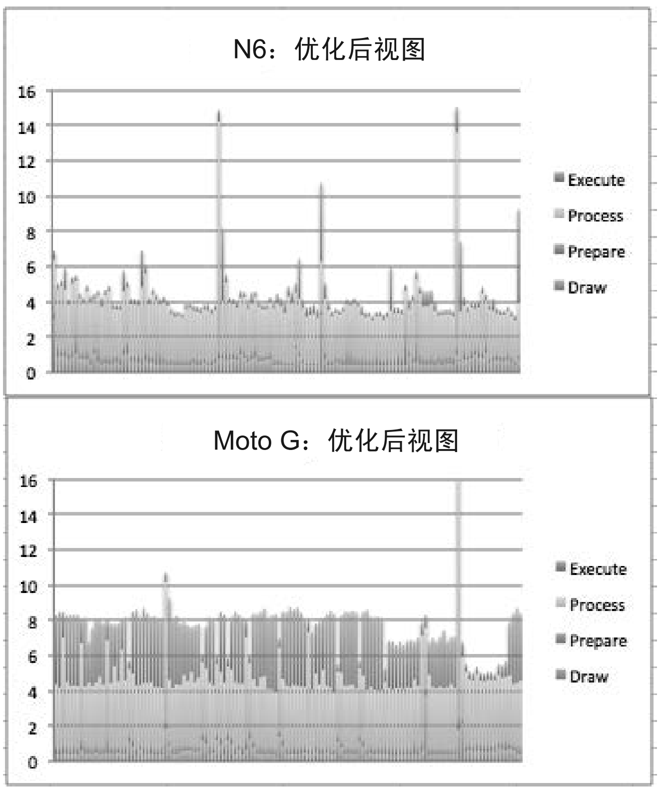


图 4-19：在 Lollipop (上) 和 KitKat (下) 上优化后视图的 GPU 概况（另见彩插）

在很大程度上来说，GPU 分析器可以帮助你发现问题。在“Is it a goat?” App 中，如果我打开斐波那契延迟（在创建视图时进行耗时的递归计算），GPU 分析器并不能看出任何卡顿，因为计算都发生在主线程而且完全阻塞了渲染。这在低端机上可能会出现 ANR 消息。[1]

注 1：当 CPU 被占用而无法处理渲染时，这个渲染时间就会超过 16 毫秒。而 GPU 分析器不会显示超过 16 毫秒的卡顿事件，因此这种情况下 GPU 分析器看不出任何卡顿。——译者注

斐波那契算法

斐波那契序列是这样一组数的集合：每个数字都是它前面两个数字的和。比如 0、1、1、2、3、5、8 等。在程序里斐波那契序列一般用来表示递归，这里我用了最低效的方式来生成斐波那契序列。

```
public class fibonacci {
        // 递归斐波那契
        public static int fib(int n){
            if (n<=0)
                return 0;
            if (n==1)
                return 1;
           return fib(n-1) + fib(n-2);
        }
```

生成一个值的计算次数呈指数增长。这样做的目的是在渲染的时候增加 CPU 的压力，这样渲染就无法得到及时处理，会出现延迟。计算 $n = 40$ 会让 App 变得迟缓（低端机上会产生崩溃）。虽然这个例子在解释渲染受阻方面有点牵强，但我们用于跟踪确认斐波那契代码的方法是有效的，能帮助我们找到致使 App 运行缓慢的代码。

Android Marshmallow里的GPU渲染

在 Android Marshmallow 中，运行 `adb shell dumpsys gfxinfo <packagename>` 命令可以看到一些新的特性，这些特性有助于实现对无卡顿渲染的追求。首先，在报告开头部分能看到 App 渲染每一帧的总结信息了：

```
**Graphics info for pid 2612 [appname]**

Stats since: 1914100487809ns
Total frames rendered: 26400
Janky frames: 5125 (19.41%)
90th percentile: 20ms
95th percentile: 32ms
99th percentile: 36ms
Number Missed Vsync: 142
Number High input latency: 11
Number Slow UI thread: 2196
Number Slow bitmap uploads: 439
Number Slow draw: 3744
```

从 App 启动开始，你可以看到一共渲染了多少帧，90% 的帧的渲染时间是在多少毫秒以内，帧最慢的渲染时间又是多少（90%、95% 和 99%）。最后五行列出了未能在 16 毫秒内完成渲染的原因。注意，这里的问题数大于卡顿的帧数，这表明一些帧会被多个问题影响。

Android Marshmallow 在 gfxinfo 库里增加了另一个好用的测试工具——`adb shell dumpsys gfxinfo <packagename>` framestats。它能够输出一张表格，该表格可以用逗号分隔每一帧事件的具体耗时。列名没有给出，但在 Android 开发者网站（https://developer.android.com/preview/testing/performance.html#timing-info）里有具体的解释。为了算出渲染中每一步的耗时，必须计算出 framestats 值之间的差异。为了简化运算，我创建了一个电子表格（https://docs.google.com/spreadsheets/d/1SppGcFmeZXe1IHPC9UZ83ChxhjxSB2aZnKn-JvkFUTM/edit?usp=sharing）用来计算感兴趣的值。当你粘贴上原始的 CSV 数据时，对每一帧的渲染来说，P-X 列的数据都是有用的（所有结果的单位都是毫秒）。

- VSYNC-Intended_VSYC（表明是否丢帧，也就是卡顿）
- 输入事件的时间（输入事件的处理时间应该小于 2 毫秒）
- 动画评估（应该小于 2 毫秒）
- 布局和测量
- `view.draw()` 方法耗时
- 同步阶段耗时（如果大于 0.4 毫秒，表示很多新的位图正被发送到 GPU 中）
- GPU 工作时间（过度绘制的时间会在这里显示）
- 绘制一帧的总时间

工作表里有两个示例数据的表，这两个表都来源于“Is it a goat?”App：goat-optim 和 goat-slowXML。图 4-20 中的 goat-slowXML 表的数据显示，一些帧（紫色）的绘制耗时超过了 16.6 毫秒。好在 VSYNC 缓冲区中有帧，才没有出现丢帧的情况（如第一列的 0 秒所示的那样）。对于缓冲区更小的设备（或是没有时间重新缓冲的 App）来说，这就有可能导致用户体验卡顿了。图表还表明，缓慢的输入事件（橙色）和动画评估事件（红色）会增加 GPU 的负担，延长全部帧的渲染时间。

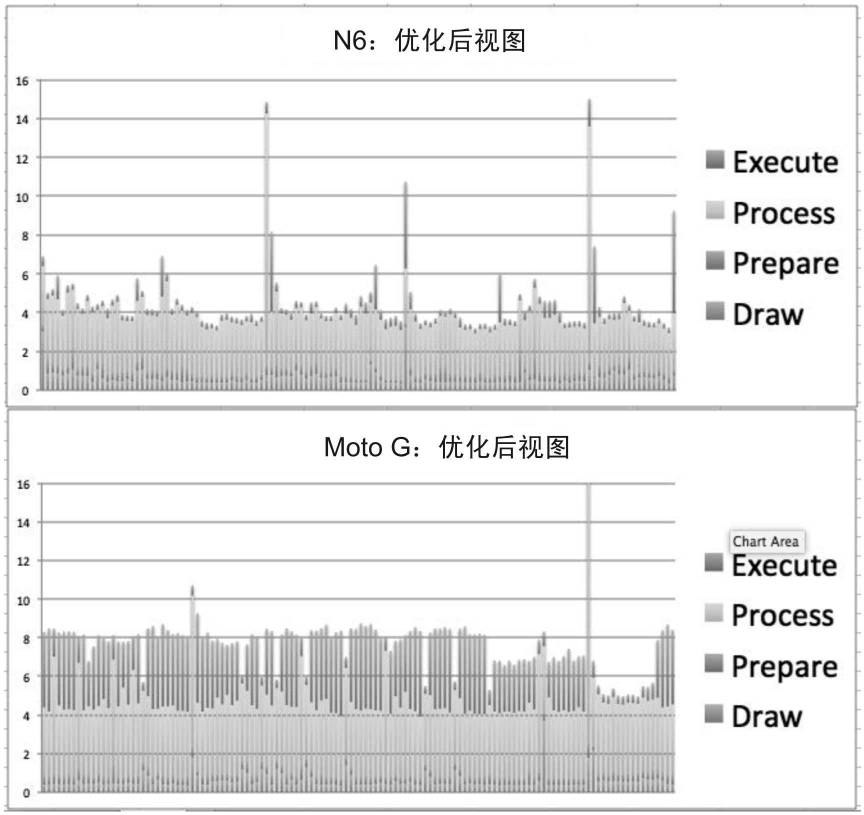


图 4-20：运行 gfxinfo framestats 命令得到的数据（另见彩插）

4.6 丢帧

很多时候，GPU 分析器不会显示超过 16 毫秒的卡顿事件，但 UI 渲染时画面的丢失或跳跃是可以觉察到的。当渲染因为 CPU 正在做其他工作而完全受阻时，就可能发生丢帧。在 Monitor 或 Android Studio 中，你可以通过 DDMS 视图查看这些日志。使用过滤器可以更清晰地看到 App 产生的这些信息。如果你认为这是 App 中的缺陷，可以查看如图 4-21 所示的警告日志信息。

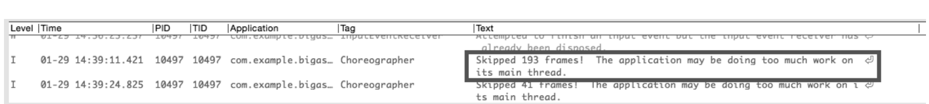


图 4-21：显示丢帧的错误日志

在第 5 章中，我们会详细讲解 CPU 是如何导致丢帧的。

4.6.1 Systrace

如果在优化了所有的视图之后，还是有丢帧现象出现的话，我们还有其他的处理办法。Systrace 工具也可以用来测量 App 性能，甚至定位问题。作为同 Jelly Bean 一起发布的“黄油计划”的一部分，Systrace 能够从内核级检测设备的运行状态。Systrace 提供了很多参数，本书的后面章节将会有所涉及。现在的关注重点是 UI 的渲染方式，以及如何使用 Systrace 调试丢帧问题。本章中出现的所有轨迹都可以在本书的 GitHub 仓库（http://github.com/dougsillars/HighPerformanceAndroidApps）里找到。

Systrace 不同于本章之前介绍的工具，它记录的是整个 Android 系统的数据，而不是某个特定的 App。因此，在它运行的时候，最好不要同时运行太多的应用，尽量减少其他应用对 Systrace 工具的干扰。在这个例子中，我们从 Android Monitor 中运行 Systrace（也可以从 Android Studio 命令行中运行）。Systrace 图标由绿色和粉红色组成（图 4-22 中的椭圆）。点击之后会弹出一个选项窗口。

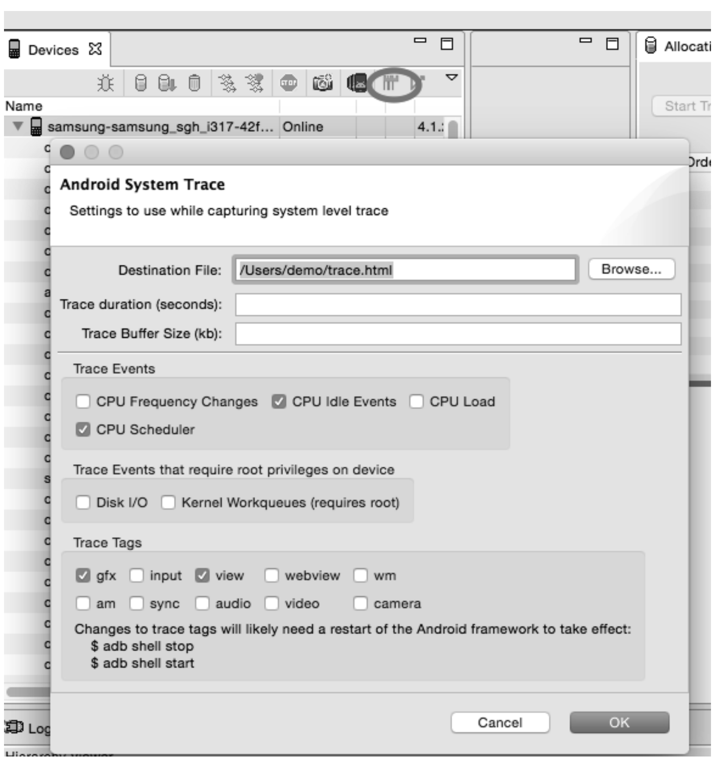


图 4-22：Systrace 的开启方式

轨迹被记录成 HTML 文件，并且能在浏览器中打开。为了研究这些屏幕上的交互，我们收集了 CPU、图形和视图的数据（如图 4-22 中的对话框所示）。不设置持续时间（默认值为 5 秒）。点击屏幕中的“OK”，Systrace 会立即开始记录你选择的参数（你最好提前完成立即开始的准备）。因为这个轨迹极其详细、数据量非常巨大，所以，最好用它来诊断某个时间段内的问题，而不是获取 App 性能的历史记录。

和 3.5.2 节类似，这些轨迹输出了大量的数据（我仅仅选择了 4 个选项）。使用鼠标可以滑动视图，WASD 键可以缩放（W，S）和左右滑动（A，D）。在轨迹顶部可以看到 CPU 的使用详情。CPU 数据下面是一些可以折叠的部分，描述了每个活动的应用。不同颜色的色条表示操作系统的不同动作，颜色长度表示动作持续的时间（放大可以看到更多细节）。选择一条色带之后，屏幕下方会显示出该项的详细信息。像 Battery Historian 和其他工具一样，这个高级视图第一眼看上去有点吓人（如图 4-23 所示）。现在我们来仔细研究一下这些信息，以便你能熟练地看懂这些文件。

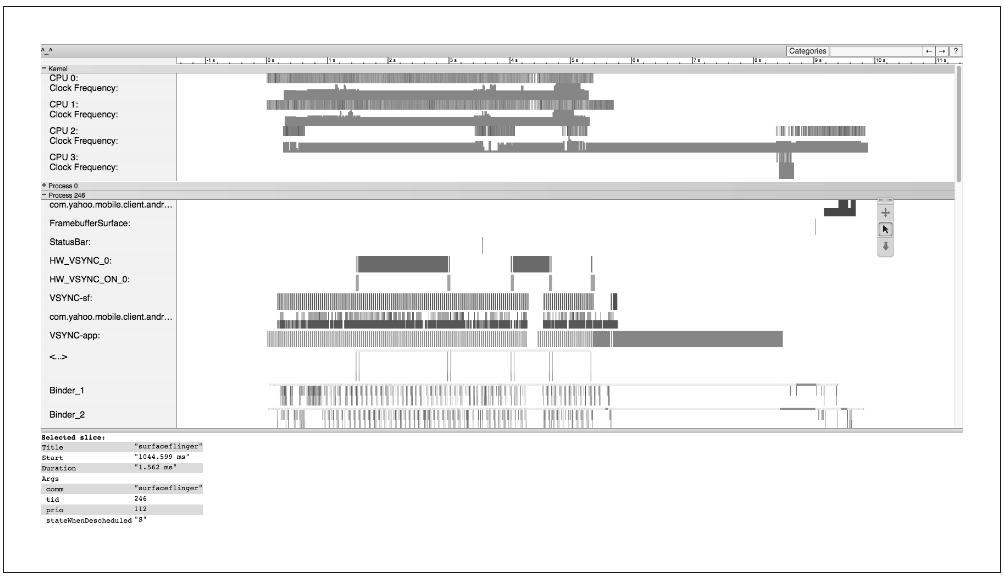


图 4-23：Systrace 的界面（Lollipop）（另见彩插）

Systrace 的演进

和 Android 生态系统一样，Systrace 的接口、展示和结果在不同的操作系统上也有一些细微的差别。

- 在运行 Jelly Bean 系统的设备上，开发者选项中有一个设置能打开记录。必须同时在电脑和 Android 设备上都开启轨迹采集。
- 随着 Android 新版本的发布，这些输出变得越来越详细，布局也发生了一些改变。
- 因为能从设备上收集到不同的信息，所以对比 Jelly Bean 和 Lollipop 的 Systrace 仍是有意义的，这两个系统中的 Systrace 存在差异。

Google 在 2015 年的 I/O 大会上发布了新版本的 Systrace，4.6.4 节中会具体讨论一些新的特性。

翻到 Systrace 结果的最后，可以看见测试过程中运行的所有进程。为了研究卡顿，需要查看 App 中有问题的绘制过程以及屏幕刷新事件。只要刷新率和绘制都正常，屏幕的渲染应该就是流畅的。但只要其中一个出现问题，就有可能导致页面渲染的卡顿。

4.6.2 Systrace Screen Painting

图 4-24 可以用来观察屏幕绘制的步骤。最上面的一行轨迹（蓝色高亮的部分）表示，VSYNC 由一些间隔均匀的宽条组成。VSYNC 是提示操作系统刷新屏幕的信号。每个色条之间都间隔 16 毫秒（也就是色条中间的白色间隔）。当 VSYNC 事件触发的时候（在每个色条的尾部），surfaceflinger（红色高亮方框中包含几种颜色的长条）会从视图缓冲区（没显示出来）里选一个视图，然后绘制到屏幕上。理想情况下，surfaceflinger 会每隔 16 毫秒触发一次（没有卡顿的情况下），因此如果出现长条空缺，则表明 surfaceflinger 丢掉了一次 VSYNC 更新——屏幕没有在规定的时间更新（这就是导致卡顿的原因）。可以看到在这段轨迹 2/3 的位置上有这样一个间隔（在绿色高亮方框中）。

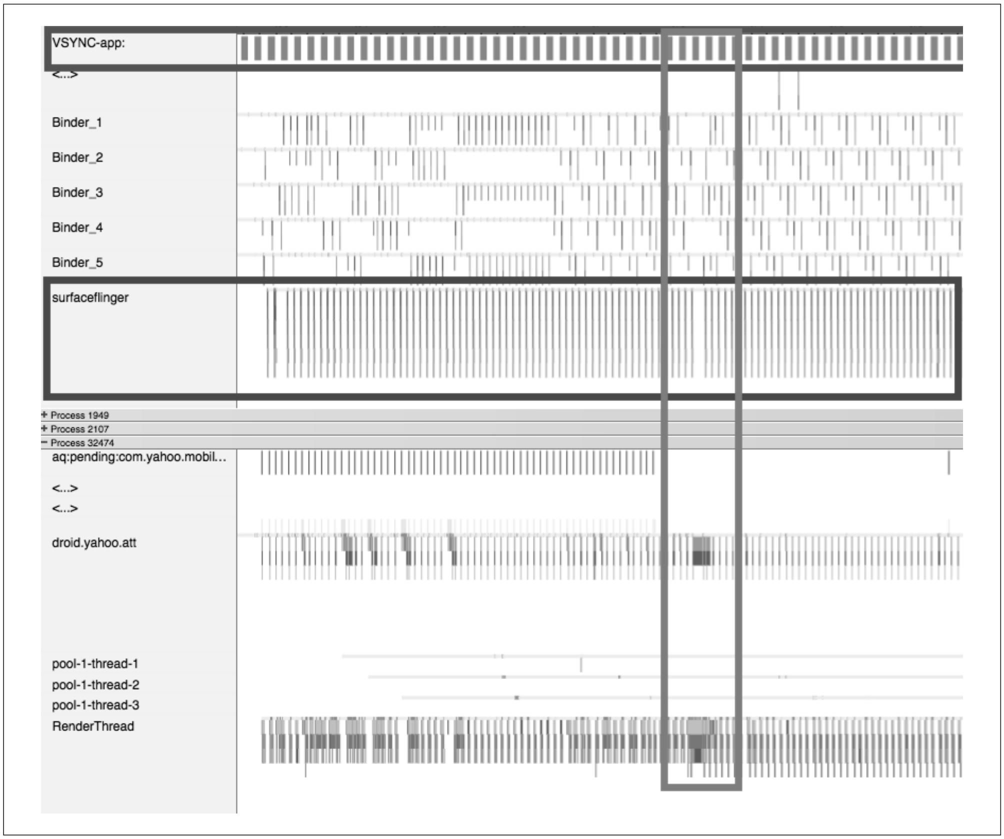


图 4-24：细看 Systrace 的卡顿部分（Lollipop）（另见彩插）

图 4-24 的下面部分显示了这个应用的详情。第二行数据（绿色和紫色的线条）表示 App 创建视图，底下的一行（绿色、蓝色和紫色）表示 RenderThread，它把渲染好的视图交给缓冲区（这个截图中没有显示出来）。注意，这些线条在同一个地方变得越来越粗，这是受 surfaceflinger 中潜在卡顿（大概在这段轨迹 1/3 的位置）的影响，这表明 App 中的某些代码引起了卡顿。不同的 App 发生卡顿的原因是不同的，但我们可以研究一些共通的原因。

这种总览方式有助于发现卡顿，但是为了更清楚地观察卡顿，我们还需要把这张图再放大一些。为了了解 Systrace 里都发生了什么，最好的方法是弄明白 Systrace 都会测量哪些量，App 流畅运行时 Systrace 的输出应该是怎么样的。一旦弄明白了 Systrace 的工作原理，那么找到问题就变得容易多了。在图 4-25 中，我将正常状态下 Systrace 中的相关线条都放在了一起（考虑到空间问题，除去很多白色间隔）。从屏幕左边，打开 droid.yahoo.com。注意，我在描述时会在轨迹文件里来回跳动到不同的位置。当绘制发生的时候——

- 红色方框：droid.yahoo.com 正在完成视图布局测量，测量结果被传给 RenderThread。
- 橙色方框：RenderThread

– 绘制帧（浅绿色）
– 刷新绘制缓冲区（灰色）
– 缓冲区出队（紫色）
– 发送给视图缓冲列表

• 黄色方框：com.yahoo.mobile.client.andr……

缓冲区里面有一些视图，线条的高度表示了缓冲区当中视图的数量。刚开始，只有一个，当新的视图加入到缓冲区中之后，高度就变成了之前的 2 倍。

• 绿色方框：VSYNC-sf 提示 surfaceflinger 有 16 毫秒的时间来渲染屏幕。里面棕色的条状表示 16 毫秒的长度。
• 蓝色方框：surfaceflinger 从队列里抓取一个视图（注意，黄色方框里的缓冲区中视图数量从 2 变为 1）。然后这个视图被发送给 GPU，这样，屏幕绘制就完成了。
• 紫色方框：VSYNC-app 提醒 App 渲染新的视图（并且发送一个 16 毫秒的定时器）。
• 一旦 VSYNC 开始，droid.yahoo.att 就不停地重复这个过程——测量视图的布局，然后发送给 RenderThread……循环往复。

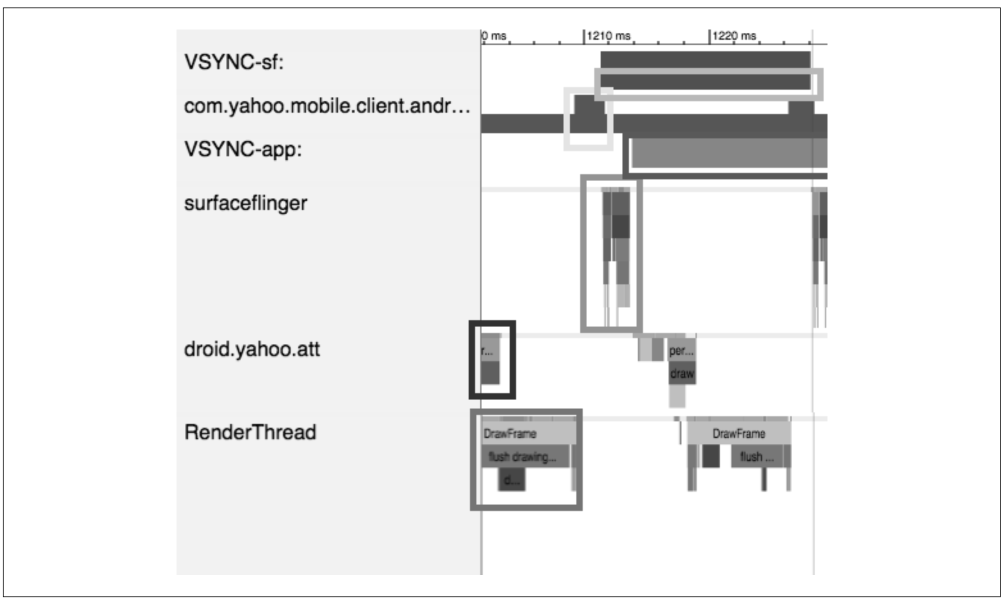


图 4-25：渲染正常的 Systrace（Lollipop）（另见彩插）

再回过头想一下，设备能在这么短的时间内流畅地渲染屏幕，确实是一件很神奇的事情。了解了渲染的过程之后，再来找一找卡顿出现的原因。

图 4-26 中可以看到操作系统层的行为。为了突出问题，我增加了一些箭头来表示 16 毫秒的间隔。右下角的方框表示 surfaceflinger 的丢帧。

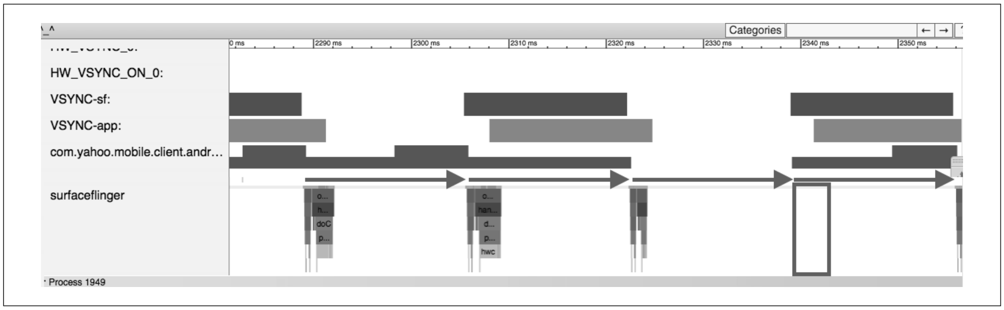


图 4-26：操作系统视图的卡顿 Systrace（Lollipop）

为什么会出现这种情况？箭头上方的一行是视图缓冲区，行的高度表示缓冲区中保存了多少帧。轨迹开始的时候，缓冲区中保存的视图数量一直是 1 或者 2。surfaceflinger 每抓取一个视图（缓冲区里的数量减少一个），App 马上又会生成一个新的视图来填充。但是，当 surfaceflinger 完成第三次动作之后，缓冲区被清空了，但是并没有从 App 里及时填充新的视图。因此，我们要从 App 层面来检查这期间到底发生了什么。

在图 4-27 中，RenderThread 在刚开始的时候发送了一个视图到缓冲区。之后 App 新建了另一个视图，渲染之后交给了缓冲区（droid.yahoo.att 测量、布局所有的视图，RenderThread 负责绘制）。不幸的是，App 没来得及创建新视图就被挂起了。在创建下一个屏幕期间，droid.yahoo.att App 在运行“performTraversals”（3 毫秒）之前，要先运行 7 毫秒的“obtainView”和 8.7 毫秒的“setupListItem”。然后 App 把数据交给 RenderThread，这一步骤相对较慢（12 毫秒）。创建这一帧总共用了近 31 毫秒（上一个只用了 6 毫秒）。当开始创建这一帧的时候，缓冲区里只有一帧的数据，但是设备在该时段需要两帧。缓冲区没有被填满，所以屏幕绘制出现了卡顿。

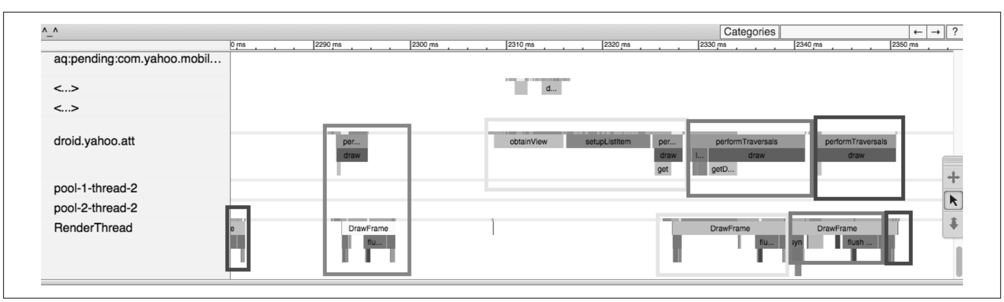


图 4-27：App 视图的卡顿 Systrace（Lollipop）

有趣的是，App 后面很快就追了上来。延迟递交的视图创建并交给缓冲区之后，后续的两帧也紧接着创建好了。通过快速地填充新帧，App 就只丢了一帧。这个轨迹结果是在 Nexus 6 上运行得到的（处理器比较快，能快速地跟上）。在 Samsung S4 Mini 和 Jelly Bean 4.2.2 上运行同样的测试，结果如图 4-28 所示。

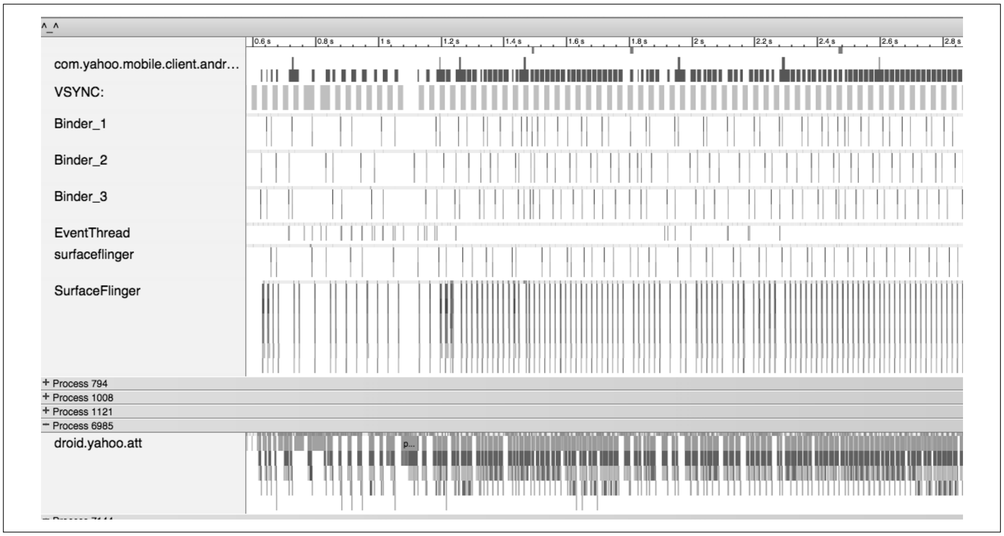


图 4-28：卡顿的 Systrace（Jelly Bean）

从缩略图上可以清晰地看到丢帧更频繁了（轨迹一开始的时候 surfacelinger 部分有很多空缺）。而且顶部第一行（视图缓冲区）里的缓冲区经常是空的（也就是说会引起卡顿），缓冲里同时有两个视图的情况非常少。在 GPU 性能比较差的设备上，App 很难像 Nexus 6 一样“追回”并填满缓冲区。

其实渲染一帧的时间可以偶尔超过 16.6 毫秒，因为缓冲区里一般都有一两帧准备好的备用视图。但是如果一行中连续出现两三帧的缓慢渲染，用户就会明显感觉到卡顿。

上面的轨迹是在运行 Jelly Bean 的手机上记录的，RenderThread 的数据归到了 droid.yahoo.att 那一行（在 Lollipop 之前，测量、绘制和布局都是在一起的）。把每一行数据合在一起之后竖条变宽。每一次调用之间的细条空白说明手机在每帧的绘制之后，只剩下少量时间处理其他任务。手机 App 的运行速度只能稍稍领先 surfacelinger 填满缓冲区的速度。如果 App 能够降低所绘制视图的复杂度——加快视图的渲染，细条中间的空白就会变得宽一点，缓冲区也就更容易被填满，在低端设备上，可以获得更多的时间去处理其他任务。

将一块区域加高亮之后，Systrace 会计算所有条状所占时间的总和，在这些数据上移动鼠标就能看到基本的统计分析了。图 4-29 中，可以看到 performTraversals（父视图的绘制命令）平均用了 13.8 毫秒，大概有 5 毫秒的波动。16 毫秒的卡顿阈值在波动的范围之内，所以我们猜测这个设备上可能有卡顿问题。

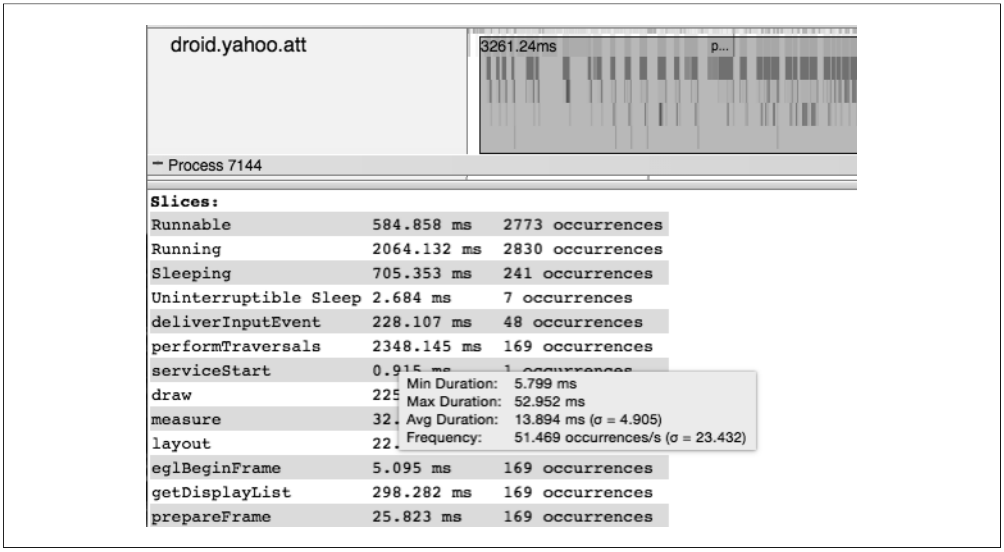


图 4-29：Systrace 数据汇总

把这块放大能看到更多的细节（见图 4-30）。每条垂直的细线表示 16 毫秒。从图中可以看出，SurfaceFlinger 大概错过了细线标记五六次。“performtraversals”线条几乎都接近 16 毫秒长（每帧之间必须执行这个方法，因此导致卡顿）。还有两个 deliverInputEvents（每个都超过了 16 毫秒）也阻碍了 App 的屏幕绘制。

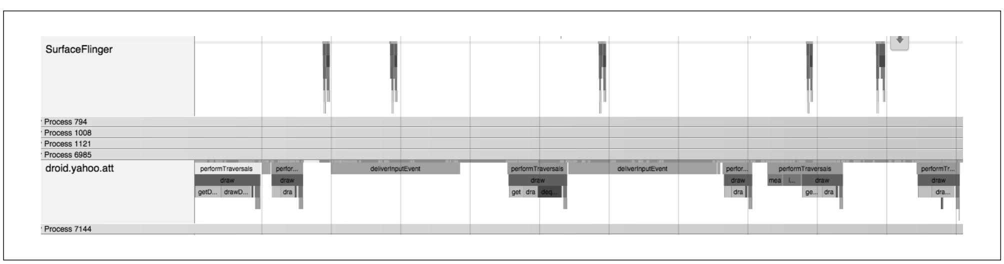


图 4-30：低端设备上的 Systrace 详情（运行 Jelly Bean）；注意帧的创建时间是怎样超过 16 毫秒造成卡顿的

那到底是什么触发了 deliverInputEvents 呢？这其实是由用户点击屏幕，强制 ListView 重绘所有视图造成的。这是 CPU 级的阻塞。接下来我们简单地看一下 CPU 阻塞是什么样子（第 5 章中会详细讲解）。

4.6.3 Systrace和CPU阻塞渲染

如果 App 频繁地发生卡顿，但是 surfaceflinger 和渲染都没有明显的异常，这时候就可以通过 Systrace 顶部的数据查看 CPU 正在被什么占用。如果能分离出某些阻碍 App 绘制的特

性或过程，那么就可以将这段代码从绘制进程中移除（通常是从主线程中移除）。在“Is it a goat?”App 有一个打开斐波那契延迟的选项。打开了这个选项，App 在渲染每一行数据的时候都会计算一个很大的斐波那契值（递归计算）。可以想象，这是一个非常缓慢而且频繁占用 CPU 的过程。因为计算阻塞了视图的渲染，所以这会导致视图创建时的丢帧和滑动时的异常卡顿。图 4-21 展示了这种情况下丢帧的日志数据，现在让我们来深入挖掘 Systrace 发现斐波那契计算的过程。

我们开始重新观察 App 正常运行时的轨迹数据。图 4-31 展示了“Is it a goat?”App 在 N6 上使用了非优化布局时的数据。

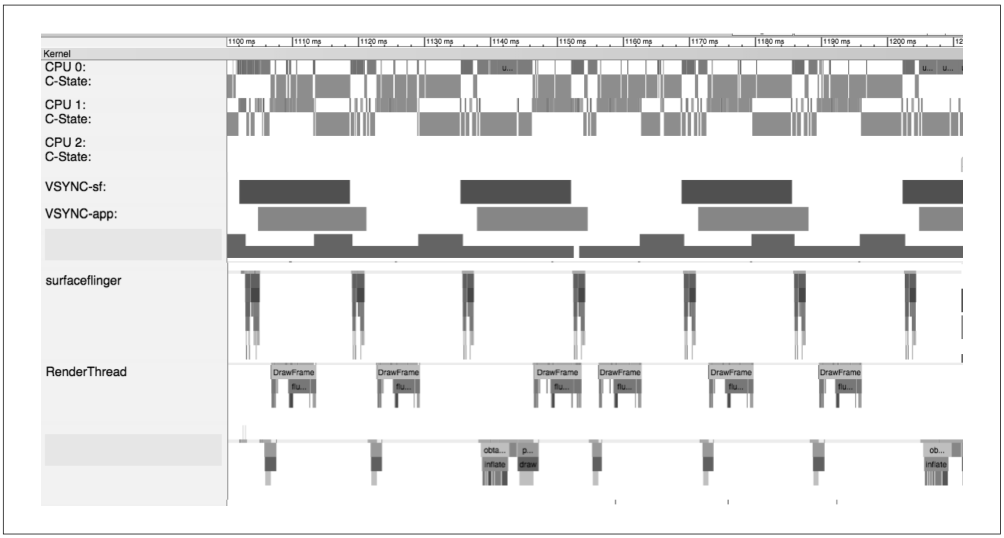


图 4-31：Systrace CPU 信息图（另见彩插）

上图是被修正过的，减少了 CPU 和 surfaceflinger 之间的许多界限。在这段轨迹中没有看到卡顿，并且 surfaceflinger 规律地每 16 毫秒运行一次（没有发生卡顿）。RenderThread 和 App 进程创建了所有的视图，并将其适当地绘制在了系统的帧缓冲区上。比较简洁模式下展示的两行 CPU 信息之后，我们发现，当 RenderThread 正在绘制布局的时候，CPU1 被蓝色活动占用（注意，我们观察的是较窄的 CPU1，不是 CPU1:C-State）。当 App 进程计算视图的大小和位置时，CPU0 显示为相应的紫色进程。整个布局的创建和绘制由两个 CPU 共同完成。注意，X 轴的时间单位为 10 毫秒，视图创建和绘制的过程都不会超过 2~4 毫秒。

绘制过程加入密集型的斐波那契计算之后，Systrace 结果就会大不相同（见图 4-32）。

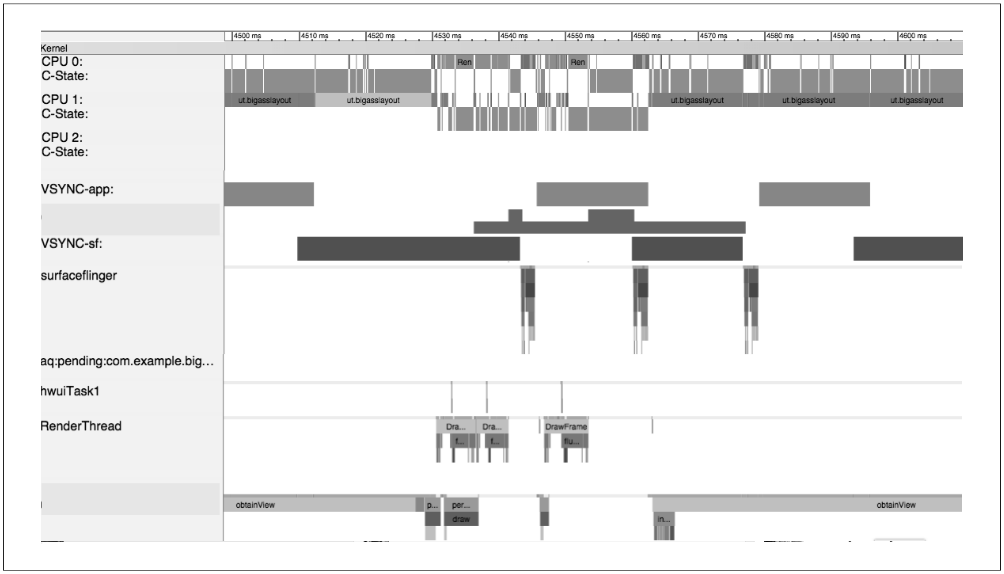


图 4-32：渲染延迟 Systrace CPU 信息图

图中 Systrace 显示了大量的卡顿。同样的 100 毫秒内，surfaceflinger 只绘制了 3 帧（没有渲染延迟的 App 绘制了 7 帧）。我们可以看到，RenderThread 绘制视图的速度一直都很快（从这段跟踪信息中可以看到，渲染线程占用的是 CPU0）。然而，当计算视图大小和位置时，大量的递归斐波那契计算导致了问题。相对于计算，App 进程在 obtainView 状态中花费了大量的时间。从图中可以看到，本不该超过 2~4 毫秒的 App 进程现在占用了 CPU1 2~17 毫秒。大量的斐波那契计算占用了 13~17 毫秒，影响了 App 视图绘制的流畅性。我们将在第 5 章中具体讨论如何诊断 CPU 的性能（以及它对渲染的影响）。

4.6.4 Systrace更新——2015年Google I/O开发者大会

Google 在 2015 年的 I/O 开发者大会上发布了新版 Systrace。在新版本中，上述的分析功能使用起来更加简便。如图 4-27 所示，我用高亮方式突出了每一帧的更新过程。在新版的 Systrace（如图 4-33 所示）当中，每一帧都用一个标有 F 的圆点表示。正常渲染的帧用绿色的圆点来表示，渲染慢（或非常慢）的帧用黄色或者红色圆点来表示。选择圆点然后按下 m，就可以凸显出一帧的数据从而更便于分析。

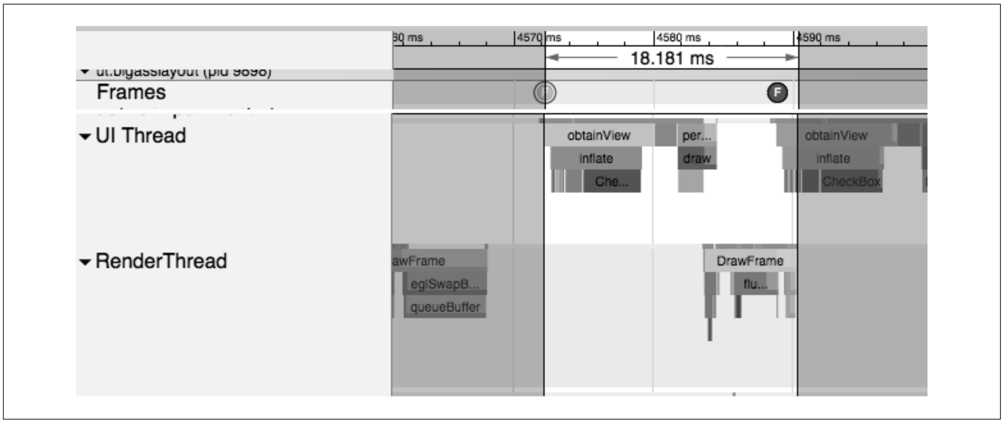


图 4-33：新版 Systrace 帧信息图

新版的 Systrace 对于正在发生的行为也有更清晰的描述。在图 4-33 中，帧标识为中间的圆点，渲染的时间为 18.181 毫秒，而过多超过 16 毫秒的帧会导致卡顿。轨迹文件下面的描述面板（如图 4-34 所示）对 App 回收 ListView 的项目进行了预警，因为比起创建新的项目，回收会让视图的生成变慢。

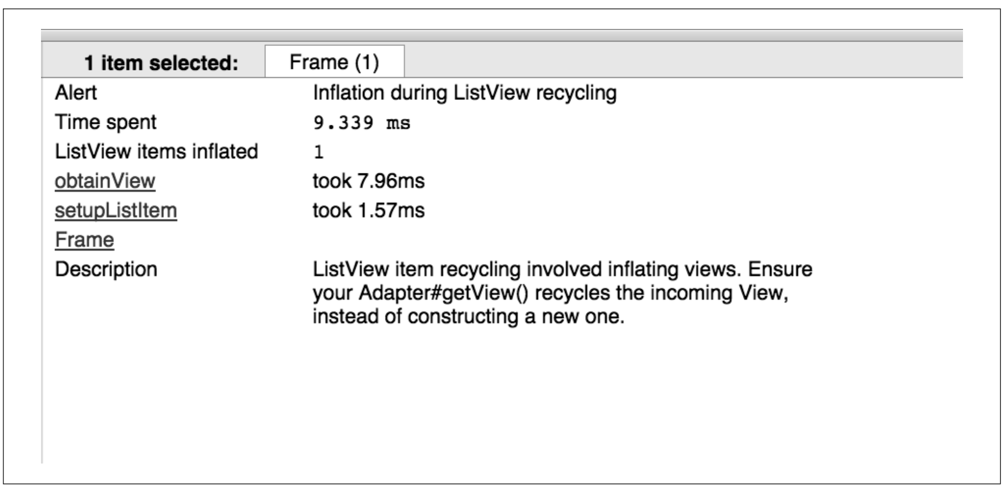


图 4-34：新版 Systrace 中帧延迟信息图

类似的警报会以相似的气泡窗或圆点的形式在 Systrace 里弹出，同时也会在屏幕右侧的 Alerts 面板列出警告（如图 4-35 所示）。

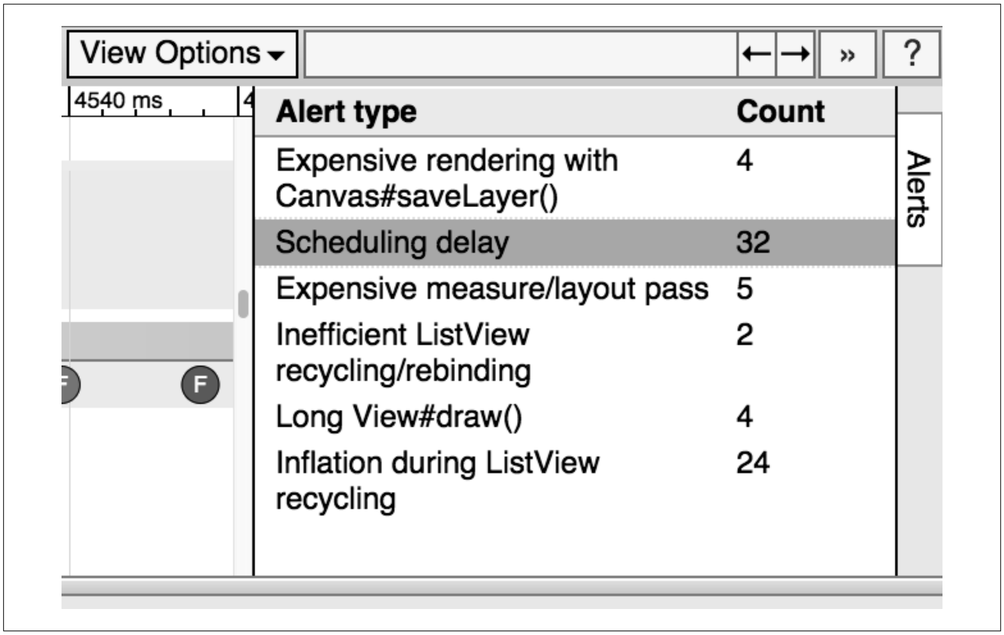


图 4-35：新版 Systrace 警告面板

这些 Systrace 新增加的特性更有利于诊断出 UI 绘制缓慢的原因。

4.6.5 第三方工具

每一个大型的芯片制造商都有自己的 GPU 评测工具，这些工具可以帮你发现更多渲染时可能遇到的瓶颈信息。针对在不同芯片组上运行的 App，这些工具提供了更多的相关信息，可以帮你就不同的 GPU 做更深度的优化。这确实远远超过了本书的研究范围，但是有必要提一下，利用这些工具甚至可以加强 GPU 的调试功能。Qualcomm、NVIDIA 和 Intel 都提供了特定的开发者工具，以测试 App 在其处理器上的 GPU 性能。

4.7 感知性能

在前面的章节中，我们讨论了怎样通过测试、发现问题和优化布局使 UI 绘制更加流畅。但是还有另一个方法能使 UI 绘制得更快：让它*看起来*更快。当然，想要 App 运行足够快的话，所有的代码、视图、过度绘制和其他可能影响 UI 绘制的相关优化工作是必不可少的。然而，即便这样做了，仍然还有几个方法可以使 App 运行得更加流畅。

人类的思维方式很有趣，通过改变等待的感知，可以让用户觉得等待的时间变短了。食杂店把廉价杂志放在收银通道两侧就是为了让客户有东西可看，从而使他们感知到的等待时间更短。如果能提供无缝对接的内容交付方式，感知到的延时会更短。这看起来似乎是提

升用户感知体验的花招，但是决定 App 流畅度的关键就在于用户的感知。这个方法实现起来有点取巧，有些时候感知性能优化反而会事与愿违，因此要进行 A/B 测试来确定优化是否有助于 App 用户的流畅体验。

4.7.1　进度条：优缺点

旋转进度条、进度条、沙漏图标，以及其他表示暂停的工具由来已久。这些工具都使 App 运行和内容过渡变得更快。比如，在 App 里加一个进度条，加载时播放动画。研究（http://www.chrisharrison.net/projects/progressbars2/ProgressBarsHarrison.pdf）表明，使用带有动画的滚动条会让用户感觉更舒服。快速旋转的进度条（http://uxmovement.com/buttons/how-to-make-progress-bars-feel-faster-to-users/）会让用户感觉等待时间变得更短了一些。

然而，延时就添加进度条并不一定是一个好主意。iOS App 开发者 Polar 注意到，他们的 App 在渲染页面视图的时候会出现一些延时。他们按照往常的习惯，在页面中加入了一个进度条来告知用户页面正在渲染，但是得到的反馈却不是很理想。反馈表明用户认为 App 变慢了，等待页面加载的时间变长了（注意：唯一的改变就是添加了进度条，App 实际上并没有变慢）。等待的标识让用户明显地感觉到他们在等待。取消进度条之后，用户感觉 App 又变快了（除了进度条并没有改动其他的代码）。通过改变用户对等待的感知，App“变得”更快了。Facebook 也曾遇到了类似的问题：他们在 iOS App 中用自定义进度条替换了标准的进度条，却让下载的感知时间变得更长了。

应该通过用户体验测试添加的进度条是否得到了预期结果。通常来讲，当延迟时间稍微有点长的时候，进度条是可以接受的，如打开新页面或者是下载网络图片时。但是如果延迟很短（1 秒之内），就应该考虑省略掉进度条。这种情况下，添加的进度条所暗示的延迟并不存在。

4.7.2　动画掩盖加载时间

点击后的白屏会让用户产生正在等待的感知。正是因为这个原因，浏览器才会在用户点击链接之后，新页面加载出来之前展示旧页面。对于移动 App 的开发者来说，可能并不希望让用户停留在旧页面，一个快速的切换动画可能就足以掩盖下一个视图出现所需要的等待时间。你可以在使用喜爱的 Android App 的时候观察一下，其中有多少 App 是从底部或边上出现的更新动画。

4.7.3　即时更新的善意谎言

如果用户更新了一个页面，页面上的数据立刻就会发生变化，即使数据还没有到达服务器端（当然，你需要确定这些数据最终会 100% 地更新到服务端）。例如，当你在 Instagram 上赞了一张照片之后，移动 App 上的视图立刻就更新为赞状态了，即使还没来得及与服

务端创建连接。他们称这种做法为“行为最优化”，几秒之内更新就会出现在网站上，并且对好友可见（如果在隧道或者信号覆盖较弱的区域可能需要几分钟），但是一定会更新，这样就不需要等待服务器更新数据之后再更新 UI 界面了。移动用户觉得他们没有义务花时间等待“确认工作”。

这样做的优势就是在网络信号不好的时候，不需要等待服务器更新数据，就能够立刻更新 UI（例如你通勤回家的火车进入隧道的时候）。Flipboard（http://t.co/lgziQm1qZ5）有离线后发送网络请求的框架，能够方便快速地更新 UI。

另一个性能技巧（和稍后上传相反）是提前上传。对于像 Instagram 这种因为需要上传大量图片而增加了主线程 UI 延迟的 App 来说，提前上传是一个好办法。Instagram 发现发帖最慢的一步是上传图片，所以在用户添加有关图片帖子的文字时，Instagram 就先将图片上传到服务器了。用户点击发帖按钮之后，只需要创建 Post 请求上传文本即可，这样用户就会觉得很快。换一种角度思考，Instagram 在遇到“是否需要添加进度条”问题的时候，通过改变 App 的架构方式杜绝了进度条。

4.7.4　提高感知性能的建议

通过优化代码和界面布局加速 App 的时候，开发者可以用秒表来测试结果。一些感知性能也可以通过秒表来测量（例如上文提到的 Instagram 的例子），但是有些（如进度条的例子）则不可以。当常规的分析测量工具不可靠时，就需要用真正的用户体验测试来确认优化效果。无论是更广范围的 A/B 测试还是可用性测试，都可以确定优化是让用户更开心还是更沮丧。

4.8　小结

Android App 的用户体验直接跟屏幕上的展示方式相关。如果 App 内容加载缓慢或者是滑动不够流畅，用户就会对 App 产生负面的体验感知。本章中，我们举例讲解了如何优化视图的树形结构和布局来加快渲染速度。我们还讲解了屏幕的过度绘制和检测过度绘制的工具。针对需要深度分析的优化问题（CPU 相关问题），Systrace 是一个强大的发现和解决卡顿问题的工具。最后，还介绍了一些优化用户感知体验和提高渲染响应速度的小技巧，例如，将 CPU 或者网络任务延后、提前或者从主线程中移除。下一章，我们将一起研究怎样通过优化和降低 App 的 CPU 占用率来提高性能。

第 5 章

内存性能

在第 4 章末尾，我们讨论了一个问题，如果 App 中的进程阻塞了 UI 线程，这将会阻止屏幕的刷新。在本章，我们将测试并更好地了解 App 是如何使用内存的。在 Android 中，内存泄露是引起程序崩溃的主要原因，运用本章讨论的工具来诊断问题可以帮助你避免泄露问题。本章的后半部分内容将涵盖降低 App 中 CPU 使用率的方法。现在我们开始讨论内存管理以及优化的诀窍。

5.1　Android内存：它是如何工作的

在讨论如何提升 Android 程序的内存利用率之前，我们需要了解一下 Android 内存管理的基础知识。有了这个坚实的基础，我们才能明白这其中的难点，并知道如何搞定它们。为了介绍一些基本术语，我们先来了解 Android 设备的一些基本信息。

大家都知道，在 Android 设备上的 Java 运行时（Dalvik 或者 ART）是一个内存管理环境。运行时通常会处理所有的内存分配和清理（垃圾回收）。它们确实能减少开发工作量，让你不用关注那么多细节，但是当你开发 App 时，就得做一些慎重的考虑，从而保证内存管理可以正确地工作。

让我们先简单了解一下 Android 程序会用到的一些内存定义。

5.1.1　共享内存与私有内存

所有的 App 都会利用这些公共的框架类、资源以及本地类库。如果每一个 App 都需要将这些单独地保存在内存中，那么可以并发运行的 App 就会减少。为了节省内存，Android

使用共享内存来保存这些资源。当分配内存给 App 时，这些共享内存将被平均分配给所有正在运行的进程。

私有内存是指只被你的 App 使用，而其他 App 不能使用的内存。因为只有你的进程可以使用这些数据，所以这些私有内存将会完全地分配给你的这个进程。

用 Zygote 作为共享内存

就像生物课上说的一样，受精卵（zygote）是在受精后创造的第一个细胞，然后它再分裂成其他多个细胞，从而变成胚胎。同样，在 Android 中，Zygote 是一个包含所有框架类、公共资源以及本地库（预装在 Zygote 内部）的进程。当 App 启动时，在加载任何你编写的代码之前，它会分支（fork）出一个 Zygote 进程（App 在系统中存活需要的一切由此开始）。这样 App 初始化就比从零开始快多了。

5.1.2 脏内存与干净内存

脏内存是指仅存在于 RAM 中的内存，如果数据从 RAM 中清除掉，App 需要重新运行才能将这些数据取回；而干净内存是指这些存储在 RAM 中的单元同样也会存储在磁盘上，如果这些数据被清理掉了，只需从设备中重新加载就可以了。

ART 和干净内存

ART 的一个主要特性是在 App 安装时进行编译，而 Dalvik 运用的是即时编译（JIT）。在运行 ART 的设备上，程序代码已经在安装的时候编译过了，并且在磁盘上也做好准备了。记住，可以从磁盘访问到的内存对象是干净的，且由于它易于恢复，当内存不足的时候可以从内存中删去。因为在当前内存中的程序代码是干净的，所以 ART 的内存管理被认为是更好的。

目前，大多数设备使用的依旧是 Dalvik 的运行时，所以内存的大多数类型还是私有脏内存（内存只能被一个 App 使用，并且此 App 也只能存储于内存中）。

5.1.3 内存清理（垃圾回收）

垃圾回收是指清理在内存中不再需要的数据对象，以便大块内存可以重新分配给新的对象。一般来说，一旦某个对象在 App 中没有一个活动的引用，就可以作为垃圾被回收了。垃圾回收器会先从根部的对象开始（它知道这些对象是活动的并且正被进程所使用），并且沿着每个引用去查找它们的关联。如果一个对象不在这个有效引用的列表中，那么它肯定不会再被使用，就可以被回收了。此时，分配给这个对象的内存空间也可以回收了。在

图 5-1 中，深色的是没有活动引用（箭头）的对象，当垃圾回收事件被触发时将会被移除。

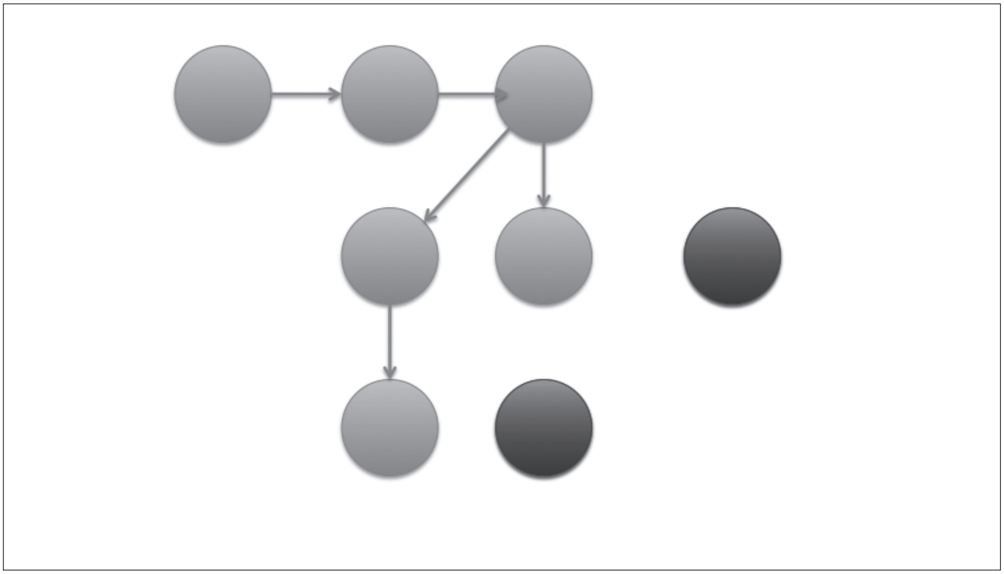


图 5-1：垃圾回收器沿着所有的引用（箭头）去查找活动的对象，将它们标注为蓝色（浅色），收集所有不是当前引用的对象并标注为红色图中的深色

1. 垃圾回收在操作系统上的改变

在 Android 上的垃圾回收随着 Android 的逐渐成熟也发生了很大的改变。在 Gingerbread 之前的版本中，设备内存很小，所以 App 的堆往往较小。垃圾回收器是一个“中断一切”的回收器，在收集垃圾的过程中，它会使所有的 CPU 进程和线程停止。这是一个全堆范围的回收，意味着回收器要遍历整个堆来寻找垃圾。对于使用内存少的 App，垃圾回收非常迅速：可能是 2~5 毫秒，你可能根本不会注意到它的存在。然而，随着设备变得越来越强大（读取更多内存），并且 App 也变得越来越大，垃圾回收花费的时间也越来越长。这些中断开始阻碍 UI，这就意味着垃圾回收器需要不断地进化。

在 Gingerbread 版本中，建立了一个并发的垃圾回收器来做局部的回收工作。尽管一个局部的垃圾回收器不能清理所有未被引用的对象，但它的速度更快（因为它并没有遍历每个集合中所有的堆）。并发垃圾回收器会和 App 一起运行，而不是阻止 App 运行。也就是说，在垃圾回收器开始和结束的时候分别有一个短暂的中断，但是它们总的中断时间不超过 5 毫秒。由于系统停止时间很短，将不再有“中断一切”的情况发生，这样也就避免了垃圾回收导致 App 卡顿。

对于在 KitKat 或者更早版本上运行的设备，垃圾回收器只是简单地做“标记清理”。旧的对象被移除，而其他所有的对象都被留在了内存当中。图 5-2 中的第一行和第二行描述了这种情况。当垃圾回收器运行时，被分配的内存（深色部分）将移除未被引用的对象，在

分配的空间中留下小块自由的内存（与被移除对象相同大小）。如果设备有一个小堆或者有很多小的集合，设备内存中可回收和可分配的内存都将被碎片化。设备可能会显示有 20MB 的自由内存，但是它不会告诉你最大块的自由内存事实上只有 1MB。如果你试着创建一个 4MB 的图片，将会遇到内存溢出问题，因为没有一块 4MB 内存可供对象使用。

在 Lollipop 版本中，Android 运行时由 Dalvik 转变为 ART，垃圾回收的效率又一次被提高了。ART 的其中一条宣言就是“垃圾回收应该是为大家提供帮助的，而不是阻碍”。垃圾回收器每一次回收只暂停一次（在 Gingerbread 版本中暂停两次），并且它们的运行速度有了显著的提升（网上有报告显示，典型的垃圾回收已经从 10 毫秒降到了 3 毫秒）。此外，在 ART 中，大型对象（比如图片）有它们自己特殊的堆，这些堆（是这些大型对象的加速垃圾回收器）专门用于简化大型对象的内存管理。

在 ART 中有许多新的垃圾回收算法，但是其中一个有趣的算法是，当一个 App 不在前台的时候，它将是一个半空间（Semi-Space）垃圾回收器。因为 App 不在前台运行，所以在内存中重写对象是安全的，可以参考图 5-2 中的第三行——没有被引用的对象被移除之后，被用过的空间被复制到内存的一块自由空间内（没有碎片）。这样就能给其他的 App 提供更多的自由内存空间。将对象移动到内存中的时候，程序必须被挂起，以避免一些并发错误。但这样就会增加 App 出现卡顿的概率，所以半空间的垃圾回收器只会在 App 在后台的时候启动。这并不完全是一个压缩的垃圾回收器，但是它对开发大块不用的空间非常有用。

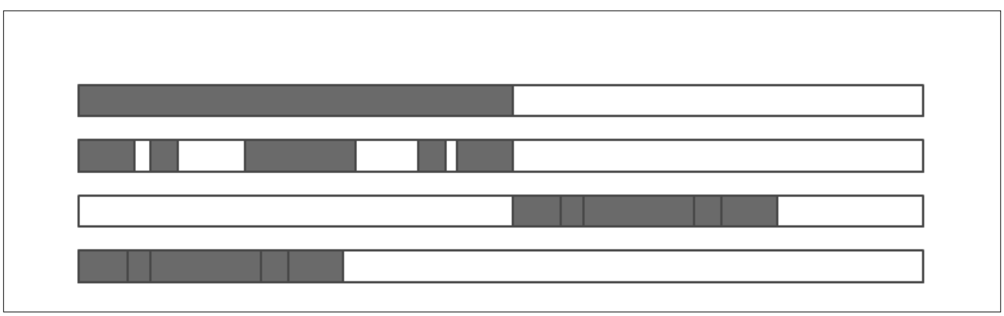


图 5-2：垃圾回收：深色表明正在用的内存，白色表明自由空间

未来：压缩垃圾回收器

2015 年，AOSP 有一个项目致力于在未来发布一个含有压缩垃圾回收器的 Android 版本。这将会进一步减少内存问题的数量，因为对象在内存的位置可以被移动以消除碎片，从而释放出大块内存。压缩垃圾回收器推动了半空间垃圾回收器的进一步发展：它在相同的内存位置重写了对象，并且没有小碎片，而不是简单地放弃一个新的内存空间。这是一个激动人心的发展，但你应该确保 C/C++ 和 NDK 代码没有引用内存地址，因为它们以后可能会因为内存被移动而有所偏差。

想知道垃圾回收器是否在运行，最简单的方法就是查看日志：

```
I/art     (10821): Explicit concurrent mark sweep GC freed
5124(199KB) AllocSpace objects, 1(16KB) LOS objects, 31% free,
34MB/50MB, paused 1.238ms total 23.656ms
```

此日志显示，进程 10821（你将会在连续分页上看到“Is it a goat？”App）正在进行一次垃圾回收。垃圾回收和此进程并发运行，阻塞 UI 1.238 毫秒（所以不大可能造成卡顿）。垃圾回收器和此 App 同时运行 23.6 毫秒。这个 App 的堆是 50MB，其中使用了 34MB，剩余 31% 未使用。垃圾回收器释放了 5124 个 AllocSpace 对象——相当于 199KB 的空间，并且从 16KB 的巨大编译空间里清理了一个占用很多内存的对象（记住，像这样的大对象在 ART 上都有自己专用的内存）。

2. 垃圾回收会在什么时候进行

当系统觉得自身需要回收内存时，垃圾回收就会进行。以下几种情况下，垃圾回收器会运行：在 App 被分配了新对象（这增加了 App 所需要的内存空间）时；新视图被创建，而旧视图无效（释放了内存中的引用）时；App 发生内存泄露，并且在内存中保存了无用的引用（阻塞了垃圾回收，同时导致了其他的内存问题）时。

下一节中介绍的工具能帮助你诊断出 App 使用内存的位置，并定位内存泄露或产生大量垃圾回收的代码，从而确保 App 在所有 Android 设备上正确地使用内存。

5.1.4　确定App使用的内存大小

现在大家已经对 App 内部内存的分配方式以及系统利用垃圾回收清理内存的方式有了大概的了解。当 App 越来越大，越来越复杂时，内存管理的指导性原则是尽可能少地使用内存。一般来说，图片是内存最大的消费者。对于网络传输来说，无论将文件压缩得多么彻底，PNG 以及 JPEG 文件都是每像素 32 比特，也就是说 100 × 100 像素的缩略图用到了 320 000 比特的内存。像这样一次性加载大量的图片，你就会明白 App 是如何占用了内存堆的 50MB~100MB 的。

在一个设备上可以使用多少内存？ `ActivityManager.getMemoryClass` 将会返回 App 可以使用的堆的最大值。如果它显示的比理想的要小，你可以减少显示内容，又或者将图片转换为较小的格式。如果 App 是内存密集型的，你可以请求 `getLargeMemoryClass()` 以获得更多内存，但是必须慎重使用它，因为在垃圾回收事件中，大的内存堆将会降低 App 的速度（因为此架构不得不通过查找更多的数据捕获无用的对象）。那么如何确定 App 是怎样使用内存的呢？

在 Nexus 6 手机上运行 `adb shell dumpsys meminfo` 命令（并且在前台运行“Is it a goat?”App），输出了以下的信息：

```
Applications Memory Usage (kB):
Uptime: 7009870 Realtime: 7218457

Total PSS by process:
   522515 kB: com.amazon.mShop.android (pid 5610 / activities)
   520153 kB: com.coffeestainstudios.goatsimulator (pid 19139 / activities)
   207397 kB: com.facebook.katana (pid 9430 / activities)
   183514 kB: com.android.systemui (pid 2111 / activities)
   141205 kB: com.example.isitagoat (pid 10821 / activities)
   113143 kB: com.google.android.googlequicksearchbox (pid 2471 / activities)
    99168 kB: system (pid 1957)
    61157 kB: com.rovio.gold_ama (pid 18842 / activities)
    58917 kB: com.amazon.kindle (pid 19331)
    49859 kB: surfaceflinger (pid 248)
    48874 kB: com.elvison.batterywidget (pid 2983)
    48270 kB: com.urbandroid.lux (pid 5656 / activities)
    35940 kB: com.facebook.orca (pid 4441)
    32541 kB: com.google.android.apps.plus (pid 20233)
    26461 kB: com.google.process.gapps (pid 2545)
    25989 kB: com.google.android.googlequicksearchbox:search (pid 2586)
    23893 kB: com.google.android.gms (pid 2610)
```

App 使用的全部内存是按比例分配的共享库占用的内存（PSS）。调用的全部内存是所有的私有内存（在某些报告里面被称为“USS-Unique Set Size”，其意为进程独自占用的物理内存，不包含共享库占用的内存）加上一定比例的共享内存。这种情况下，后台仍然运行着一些 App，这些 App 使用的内存比“Is it a goat?”App 使用的 141 205KB 内存更多。注意，“It is a goat?”App 的 PID 是 10 821，这个标识符是 Android 用来标识这个 App 的。

在报告的后面将进一步地分析内存的使用情况。首先，可以看到系统使用了多少内存来渲染视图（是否还记得 4.6.1 节中提到的服务端？）。多媒体服务器同样使用了很多内存，但是本地内存中还有几百个小的进程（出于文章篇幅的考虑，上面的表格只显示部分内容）。除了本地和系统，还有一些 App 使用内存，这些 App 拥有一些“永久”的进程，这些进程一直运行于设备上——系统 UI、NFC 以及手机上。

下一节中，我们将会了解到 App 究竟在设备何处运行。你可以看到“Is it a goat?”App 在前台运行。可见和可感知的 App 会在屏幕上出现（比如，作为通知在状态栏上出现），或者作为输出视频的 App（如 lux App）。

A 服务、B 服务以及缓存的 App 都是在后台运行的 App，并且有内存分配给它们的线程使用。它们要么是在过去运行过，在将来内存紧张的时候会被清理掉，要么确实是偶然在后台运行：

```
Total PSS by OOM adjustment:
   105030 kB: Native
               49859 kB: surfaceflinger (pid 248)
               17010 kB: mediaserver (pid 1539)
                4785 kB: rild (pid 1537)
                3555 kB: logd (pid 243)
```

```
            3494 kB: mm-qcamera-daemon (pid 1553)
            3466 kB: zygote (pid 1546)
            2405 kB: gsiff_daemon (pid 1549)
            1669 kB: sensors.qcom (pid 250)
            1610 kB: drmserver (pid 1538)
            1407 kB: thermal-engine (pid 1545)
            1260 kB: ks (pid 768)
            1188 kB: netd (pid 1535)
            1128 kB: sdcard (pid 1550)
            1072 kB: wpa_supplicant (pid 2188)
                            //plus a lot more

  99168 kB: System
           99168 kB: system (pid 1957)
 221364 kB: Persistent
          183514 kB: com.android.systemui (pid 2111 / activities)
           16764 kB: com.android.nfc (pid 2418)
           16231 kB: com.android.phone (pid 2442)

            4855 kB: com.android.server.telecom (pid 2392)
 141205 kB: Foreground
          141205 kB: com.example.isitagoat  (pid 10821 / activities)
  60554 kB: Visible
           26461 kB: com.google.process.gapps (pid 2545)
           19900 kB: com.google.process.location (pid 2917)
            9669 kB: com.google.android.inputmethod.latin (pid 2304)
            4524 kB: com...googlequicksearchbox:interactor (pid 2270)
  97144 kB: Perceptible
           48874 kB: com.elvison.batterywidget (pid 2983)
           48270 kB: com.urbandroid.lux (pid 5656 / activities)
  16113 kB: A Services
            8538 kB: com.google.android.gms.wearable (pid 3056)
            7575 kB: android.process.media (pid 29108)
 113143 kB: Home
          113143 kB: com...googlequicksearchbox (pid 2471 / activities)
 859266 kB: B Services
          522515 kB: com.amazon.mShop.android (pid 5610 / activities)
          207397 kB: com.facebook.katana (pid 9430 / activities)
           58917 kB: com.amazon.kindle (pid 19331)
           35940 kB: com.facebook.orca (pid 4441)
           25989 kB: com...googlequicksearchbox:search (pid 2586)
            4317 kB: org.simalliance.openmobileapi.service:remote (pid 4903)
            4191 kB: com.android.sdm.plugins.sprintdm (pid 14923)
 809634 kB: Cached
          520153 kB: com...goatsimulator (pid 19139 / activities)
           61157 kB: com.rovio.gold_ama (pid 18842 / activities)
           32541 kB: com.google.android.apps.plus (pid 20233)
           23893 kB: com.google.android.gms (pid 2610)
           22116 kB: com.levelup.touiteur (pid 19038)
           18309 kB: com.mobileiron (pid 26851)
           17872 kB: com.linkedin.android (pid 27259)
           15763 kB: com.amazon.mShop.android.shopping (pid 24968)
           15177 kB: com.google.android.apps.magazines (pid 26772)
           14078 kB: android.process.acore (pid 26874)
           13740 kB: com.google.android.music:main (pid 24911)
```

```
              13280 kB: com.android.mi.email (pid 26748)
              12072 kB: com.yahoo.mobile.client.android.mail.att:
  com.yahoo.snp.service (pid 26904)
              10377 kB: com.alphonso.pulse (pid 26441)
               5504 kB: com.android.chrome (pid 24943)
               4823 kB: com.google.android.deskclock (pid 28469)
               4717 kB: com.android.cellbroadcastreceiver (pid 20193)
               4062 kB: com.android.defcontainer (pid 28116)
```

最后，报告分析了所有不同类型内存的使用情况，并对比了空闲的 RAM 和使用过的 RAM。在 Nexus 6 的机器上，可以很清晰地看到缓存的 App 很可能仍然驻留在内存中，同时接近 50% 的 RAM 仍旧是空闲的：

```
Total PSS by category:
   789741 kB: Unknown
   501966 kB: Dalvik
   463460 kB: GL
   225937 kB: Other dev
   204576 kB: Graphics
    74916 kB: Ashmem
    63123 kB: .so mmap
    56944 kB: .dex mmap
    41319 kB: image mmap
    36328 kB: Dalvik Other
    22039 kB: code mmap
    20906 kB: .apk mmap
    12460 kB: Stack
     4998 kB: Other mmap
     3716 kB: .jar mmap
      112 kB: Cursor
       56 kB: .ttf mmap
       24 kB: Native
        0 kB: Memtrack

Total RAM: 3041412 kB (status normal)
 Free RAM: 1465830 kB (809634 cached pss + 450524 cached + 205672 free)
 Used RAM: 1967459 kB (1712987 used pss + 71340 buffers +
                       101780 shmem + 81352 slab)

 Lost RAM: -391877 kB
   Tuning: 256 (large 512), oom 325000 kB, restore limit 108333 kB (high-end-gfx)
```

这篇报告很好地概述了设备上的内存使用情况，但是你可能对 App 的细节信息更感兴趣。可以在 `meminfo` 命令中添加对应 App 的 PID 来了解 App 正在使用的 RAM 的更多信息：

```
adb shell dumpsys meminfo 10821
Applications Memory Usage (kB):
Uptime: 10475753 Realtime: 10684340

** MEMINFO in pid 10821 [com.example.isitagoat] **
                 Pss  Private  Private  Swapped     Heap     Heap     Heap
               Total    Dirty    Clean    Dirty     Size    Alloc     Free
```

```
                  ------    ------    ------    ------    ------    ------    ------
     Native Heap       0         0         0         0     13752     13752     29255
     Dalvik Heap   13639     13080         0         0     42782     34636      8146
    Dalvik Other     556       556         0         0
           Stack     132       132         0         0
       Other dev    6622      6592         4         0
        .so mmap    1082       164        60         0
       .apk mmap      52         0         0         0
       .ttf mmap       0         0         0         0
       .dex mmap       8         0         8         0
       code mmap     471         0        16         0
      image mmap     832       532         0         0
      Other mmap      17         4         0         0
        Graphics   66784     66784         0         0
              GL   26356     26356         0         0
         Unknown   11799     11736         0         0
           TOTAL  128350    125936        88         0     56534     48388     37401

    Objects
                  Views:      121         ViewRootImpl:        1
            AppContexts:        3           Activities:        1
                 Assets:        2        AssetManagers:        2
          Local Binders:        8        Proxy Binders:       16
       Death Recipients:        0
        OpenSSL Sockets:        0

    SQL
            MEMORY_USED:        0
     PAGECACHE_OVERFLOW:        0          MALLOC_SIZE:        0
```

这是“Is it a goat?”App 在 Nexus 6 手机前台运行时，其内存使用情况。让我们一起分析一下上面的表格究竟在说明什么事情。我们只关心前两列的数据：所有正在使用的内存（PSS = 共享内存 + 私有内存）以及私有的脏内存（只在 App 上使用，并且不会在磁盘上存储的内存）。注意，内存分配的绝大部分是私有的脏数据：

- 13MB 来自于 Dalvik（这是假设的，也可以是 ART，没有什么区别）
- 66.7MB 将分配给图形
- 26MB 将用于 GL 渲染
- 11.8MB 未知
- 6.6MB 用于“其他设备”
- 较小的分配（大部分是一些较小的共享资源）

内存总共使用了 128MB。在 ART 中，图形会存储在主堆中一个新的“大对象空间”。该空间会更好地促进垃圾回收，减少图像碎片（图像是内存中最大的对象），使堆变得越来越小。

第二张表格是正在使用的内存的信息：视图数量、资源数量，以及活动的数量。如果出于某种原因，这些数量比预期高出许多，那么就等待一秒钟，然后重新运行 `meminfo` 命令。

垃圾回收器有可能正在清理最近无效的视图。如果这些视图没有被清理（或者当 App 被使用时，视图的数量增加），很可能是出现了内存问题。表格还展示了分配给数据库的内存，以及用于 App 的其他文件。

5.1.5 procstats

`meminfo` 命令能在瞬间给出大量的信息。内存泄露一般在 App 运行一段时间后发生，与之相关的多个 `meminfo` 报告会变得迟缓，在这种情况下排查问题是很困难的。KitKat 版本引进了 procstats 以了解在后台运行 App 一段时间后的内存使用量。在设置 → 开发者选项 → 进程统计中，可以看到一个可视的设备内存使用量的界面（默认统计时间是最近的 3 小时，但是可以改为 6、12 或者 24 小时）。在图 5-3 中，屏幕的顶部显示设备内存的当前使用状态，底下的长条显示一段时间内内存的使用情况（绿色、黄色以及红色对应的是内存问题的严重程度）。点击长条可以看到更多的细节：每个内存状态所用的时间，以及内存当前是如何分配的。如果想看在前台运行或者缓存的 App 的内存使用情况，可以在设置菜单中改变状态的类型。

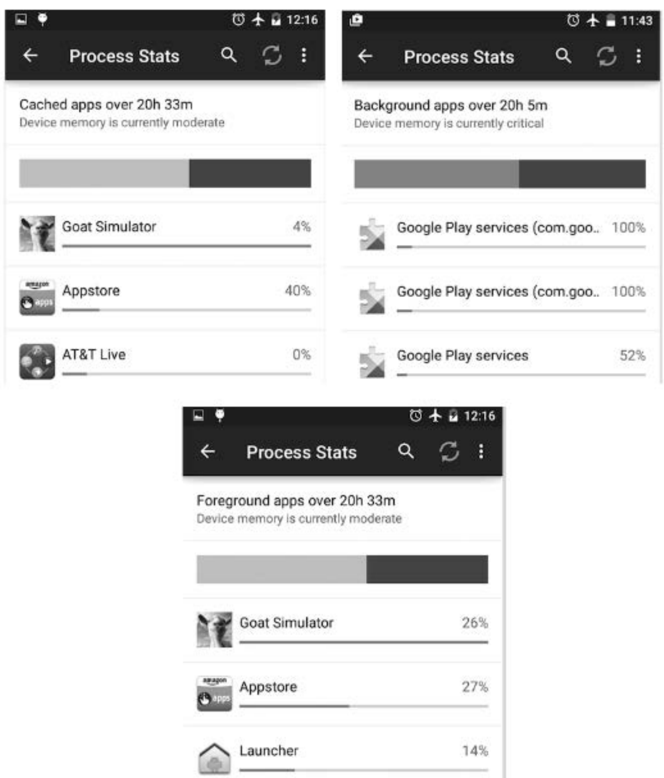


图 5-3：procstats 展示的 App 内存的使用情况，状态分别是缓存（左上）、后台（右上）、前台（底部）

任何运行状态的 App 会同其活跃时间的百分比一起被罗列出来，并且会和各自所用的平均时间作比较（同样分为前台、后台以及缓存的情况）。点击 App 可以看到它使用内存的详细信息，以及父菜单状态下的 RAM 和运行时。如果想看到 App 在其他状态下的表现（见图 5-4，前台和缓存状态），必须返回主菜单改变状态，然后再重新选择这个 App。由于观察一个 App 需要反复选择这些菜单，通常会用命令行版本的 `adb dumpsys procstats` 获取数据表。

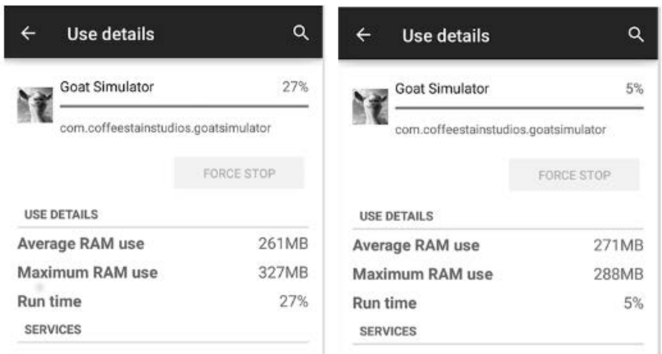


图 5-4：App 的 Procstats：Lollipop 版本下 App 处于前台（左）和缓存（右）的情况

比较图 5-4 与命令行产生的大量信息。转储包含了过去 24 小时、3 小时以及当前所有状态下的统计数据。受文章篇幅局限，只向大家展示最近 3 小时的数据（长时间的数据与此类似）。第一组数据分析了屏幕开启（SOn）和关闭（SOff）情况下系统所用的内存情况（见例 5-1）。

例 5-1：procstats 系统信息

```
$ adb shell dumpsys procstats com.coffeestainstudios.goatsimulator

AGGREGATED OVER LAST 3 HOURS:
System memory usage:
  SOff/Norm: 1 samples:
                        //与屏幕开启类似
       Mod : 1 samples:
                        //与屏幕开启类似
       Crit: 1 samples:
                        //与屏幕开启类似
  SOn /Norm: 3 samples:
        Cached: 304MB min, 317MB avg, 336MB max
        Free: 32MB min, 44MB avg, 57MB max
        ZRam: 0.00 min, 0.00 avg, 0.00 max
        Kernel: 41MB min, 46MB avg, 50MB max
        Native: 45MB min, 49MB avg, 50MB max
       Mod : 1 samples:
        Cached: 182MB min, 182MB avg, 182MB max
        Free: 24MB min, 24MB avg, 24MB max
```

```
        ZRam: 0.00 min, 0.00 avg, 0.00 max
        Kernel: 41MB min, 41MB avg, 41MB max
        Native: 46MB min, 46MB avg, 46MB max

      Low : 3 samples:
       Cached: 186MB min, 226MB avg, 287MB max
       Free: 19MB min, 104MB avg, 269MB max
       ZRam: 0.00 min, 0.00 avg, 0.00 max
       Kernel: 38MB min, 38MB avg, 39MB max
       Native: 46MB min, 47MB avg, 47MB max
      Crit: 5 samples:
       Cached: 146MB min, 179MB avg, 247MB max
       Free: 16MB min, 57MB avg, 130MB max
       ZRam: 0.00 min, 0.00 avg, 0.00 max
       Kernel: 38MB min, 40MB avg, 45MB max
       Native: 43MB min, 46MB avg, 49MB max
<big snip>

Summary:
<snip>
Run time Stats:
  SOff/Norm: +1h12m4s41ms
       Mod : +3m0s428ms
       Low : +1s954ms
       Crit: +1m7s324ms
   SOn/Norm: +27m26s70ms
       Mod : +6m9s749ms
       Low : +7m58s126ms
       Crit: +23m52s476ms
      TOTAL: +2h21m40s168ms
```

当设备的内存由正常水平变为中低水平和临界水平的时候，缓存内存将被清理以保证活动的进程可以正常运行（当屏幕开启，正常水平转为临界水平时，平均缓存内存从 304MB 降至 146MB）。在转储底部的总结分析了不同内存状态下的 3 小时数据。它显示了设备在 3 小时中运行了 2 小时 21 分钟的例子。当屏幕关闭时，设备基本处于正常的内存状态；当屏幕开启时，设备超过一半的时间都处于低内存水平或临界水平。

那究竟是什么原因使设备进入低内存的状态呢？通过研究 Goat Simulator 的报告（下方），内存问题发生的原因便逐渐清晰了。第一张表格显示了进程和一系列与 MB 相关的数据。App 有 15% 的时间（2.5% 的时间用于最后活动的进程）在前台运行（TOP）。报告中内存数据以 MB 为单位，并且有一定的格式（总内存：低水平 – 平均水平 – 高水平 / 私有内存：低水平 – 平均水平 – 高水平）：

```
Per-Package Stats:
  * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
      * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
               TOTAL: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
                 Top: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
          (Last Act): 2.5% (260MB-273MB-292MB/256MB-268MB-287MB over 3)
            (Cached): 2.7% (268MB-271MB-288MB/263MB-267MB-284MB over 7)
```

接下来的部分和总系统内存表相似，但是只分析了 Goat Simulator 的进程。首先，这部分显示了屏幕打开和关闭时该进程在不同内存状态下运行的时间。在例 5-1 的“运行时间统计”中可以看出：当屏幕关闭时，设备有 1 分 7 秒的时间处于内存临界状态。下表中 Goat Sinulator 在临界状态下所运行的时间，即活动的（表 TOP）46 秒（45 秒 940 毫秒）和最后活动的（表中 LastAct）21 秒（21 秒 384 毫秒）相加所得的值：67 秒 [1]。这同样适用于屏幕开启的情况，设备处于临界水平将近 24 分钟 [2]，Goat Simulator 在前台运行和结束时有 23 分钟处于临界状态，即 top（表中 TOP）和最后的登录时间（表中 LastAct）相加所得的值 [3]。这说明设备的内存状态可能和这个 App 的内存使用是有关联的。

在例 5-2 中，我们将追加一个内存使用以分析这个 App 的不同状态。

例 5-2：procstats App 信息

```
Multi-Package Common Processes:
  * com.coffeestainstudios.goatsimulator / u0a82 (16 entries):
        SOff/Norm/LastAct: +1m32s937ms
             Mod /LastAct: +76ms
             Low /LastAct: +1s870ms
             Crit/Top    : +45s940ms
                  LastAct: +21s384ms
        SOn /Norm/Top    : +20s540ms
                  LastAct: +9s755ms
             Mod /Top    : +8s70ms
                  LastAct: +11s263ms
                  CchAct : +1s571ms
             Low /Top    : +21s335ms
                  LastAct: +10s802ms
                  CchAct : +3m31s584ms
             Crit/Top    : +20m0s324ms
                  LastAct: +2m55s742ms
                  CchAct : +18s8ms
                  TOTAL  : +30m51s201ms
        PSS/USS (10 entries):
          SOff/Crit/Top    : 1 samples 275MB 275MB 275MB / 270MB 270MB 270MB
                    LastAct: 1 samples 266MB 266MB 266MB / 261MB 261MB 261MB
          SOn /Norm/Top    : 1 samples 136MB 136MB 136MB / 127MB 127MB 127MB
               Mod /Top    : 1 samples 174MB 174MB 174MB / 167MB 167MB 167MB
                    LastAct: 1 samples 251MB 251MB 251MB / 248MB 248MB 248MB
               Low /Top    : 2 samples 155MB 201MB 247MB / 150MB 196MB 242MB
                    LastAct: 1 samples 260MB 260MB 260MB / 256MB 256MB 256MB
                    CchAct : 7 samples 268MB 271MB 288MB / 263MB 267MB 284MB
               Crit/Top    : 18 samples 119MB 279MB 327MB / 113MB 273MB 321MB
                    LastAct: 1 samples 292MB 292MB 292MB / 287MB 287MB 287MB

Summary:
```

注 1：例 5-2 中 SOff Crit 一栏中 Top 和 LastAct 的值。——译者注

注 2：例 5-1 中运行时间统计中的：Crit: +23 分 52 秒 476 毫秒。——译者注

注 3：例 5-2 中 SOn Crit 一栏中的 Top 和 LastAct：20 分 0 秒 324 毫秒 +2 分 55 秒 742 毫秒 =23 分。——译者注

```
  * com.coffeestainstudios.goatsimulator / u0a82 / v915134:
           TOTAL: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
             Top: 15% (119MB-261MB-327MB/113MB-255MB-321MB over 23)
      (Last Act): 3.8% (251MB-267MB-292MB/248MB-263MB-287MB over 4)
        (Cached): 2.7% (268MB-271MB-288MB/263MB-267MB-284MB over 7)

<snip>

          Start time: 2015-01-23 15:38:18
  Total elapsed time: +21h33m58s23ms (partial) libart.so

          Start time: 2015-01-24 11:48:20
  Total elapsed time: +1h23m56s121ms (partial) libart.so
```

当在内存中注入一个对象时，Android 系统就会为此对象分配内存，对象不再被使用的时候，会利用垃圾回收机制回收该对象的内存。在 5.1.3 节中，我们已经讨论过内存是如何进行清理工作的，以及当 App 使用过多内存时，内存回收机制又是如何决策的。procstats 命令提供了设备内存状态的相关信息。在 5.1.6 节中，我们将会介绍当内存有限时，如何利用 App 的警告来确保 App 可以持续运行。

5.1.6 Android内存警告

Android 系统为每一个 App 分配可用的内存堆，并且负责执行垃圾回收（从内存中移除旧的内容）的任务。在 5.1.5 节中，可以看到 procstats 的报表显示内存正处于临界水平。处于运行（或缓存）状态的 App 会释放内存以防止自己被清理掉。onTrimMemory 会显示 App 被缓存在什么位置，以及如何释放内存，从而保证 App 不会被移除。如果 App 正在运行，并存在内存问题，onTrimMemory 会发出以下警告：

- TRIM_MEMORY_RUNNING_MODERATE

  首先发出 TRIM_MEMORY_RUNNING_MODERATE 警告。

- TRIM_MEMORY_RUNNING_LOW

  如果继续执行，将会发出 TRIM_MEMORY_RUNNING_LOW 警告，就像是黄灯警示。这时系统会开始释放资源[4]来提高系统性能。

- TRIM_MEMORY_RUNNING_CRITICAL

  如果仍然继续执行并且没有释放资源，将会发出红灯警告：TRIM_MEMORY_RUNNING_CRITICAL。此时，系统会结束后台进程以获取更多的内存。同时，这将降低 App 的性能。

- TRIM_MEMORY_UI_HIDDEN

  当回调 TRIM_MEMORY_UI_HIDDEN 时，App 刚从前台转为后台，这是释放大量 UI 资源的大

注 4：释放无用资源。——译者注

好时机。此时 App 在缓存的 App 列表中。如果有问题，此 App 的进程将会被结束。作为一个后台程序，尽可能多地释放资源，这样的恢复会比纯粹的重启[5]更加快速。其中有 3 个级别：

- `TRIM_MEMORY_BACKGROUND`

  App 处于列表[6]中，但是是接近尾部的位置[7]。

- `TRIM_MEMORY_MODERATE`

  App 处于列表的中部。[8]

- `TRIM_MEMORY_COMPLETE`

  这是“下一个被结束的就是此 App”的警告。[9]

在例 5-2（在“Summary”上面的两行）中，可以看到内存处于临界的时间，以及当 App 从“屏幕上”最上层出发，从前台运行到后台运行，以及“最后活动的状态”时，最大内存使用量从 327MB 降到 292MB。

5.2 Java中的内存管理/泄露

对于内存管理来说，第一准则永远是减少内存中存储的信息量。通过减少内存使用，内存中的对象越来越少，被回收的对象也相应减少，内存相关的问题也越来越少，从而垃圾回收的速度也越来越快。在本章的后面会介绍多余的对象是怎么影响内存使用和 App 性能的。

尽管在 Android 运行时中内存是被管理的，但是开发者一定依旧担心内存是如何被使用的。当内存中存有不必要的对象时，Android App 就有可能发生内存泄露。活动之间有意外的引用，或者其他链接导致垃圾回收不停地回收对象，都可能引起内存泄漏。这种意外的引用在低内存设备中会导致内存不足，所以找到内存泄露并解决它们非常关键。如果在 App 中发现内存问题（或者在本章讨论的工具中看到出乎预料的结果），那么有可能发生了内存泄露，一定要追根究底，发现并消除这个泄露。

5.3 追踪内存泄露的工具

前文中所说的 `meminfo` 工具在确定是否有内存泄露方面很有用。如果来自 `meminfo` 和 Process Stats 工具的结果不尽如人意（后台运行占用超乎预期的内存量或者内存使用意外地增长），还有更多工具可以帮助你发现到底是何处存在内存泄露。每个内存泄露都有其

注 5：App 由于内存不足被结束后的重启。——译者注
注 6：缓存应用列表。——译者注
注 7：意思是 App 被结束的风险不高。——译者注
注 8：App 有被结束的风险。——译者注
注 9：此时应该释放所有非关键的资源从而恢复应用的状态。——译者注

独特性，对于每个代码库来说，发现它们的路径也是不同的，但是下面的这些例子能给你指出一个正确的开始方向。

5.3.1 Heap Dump

在内存中，App 到底使用了多少数据？在 App 运行的过程中，什么类型的文件会被分配到内存当中？在 DDMS 监控器中有一种非常好的工具——Heap Dump，可以用来解决以上这些问题。想要激活 Heap Dump，先选择 App，然后启动 Update Heap 按钮——该按钮为圆柱体，并且装有一半的绿色液体（难道是 Android 的血液？）。这将填充屏幕右侧的菜单和按钮。想要知道当前 App 用了多少内存，只需要点击 Cause GC 按钮。这么做会强制 App 进行垃圾回收，清理一部分文件。然后根据类别和大小，计算好剩余的文件，并将它们罗列显示在 Heap 工具中（在 Nexus 7 上的 Android 5.0.2 版本）。见图 5-5。

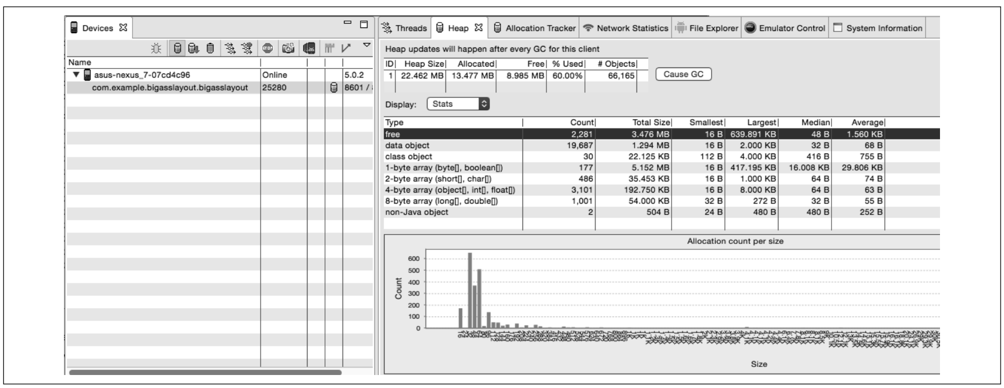
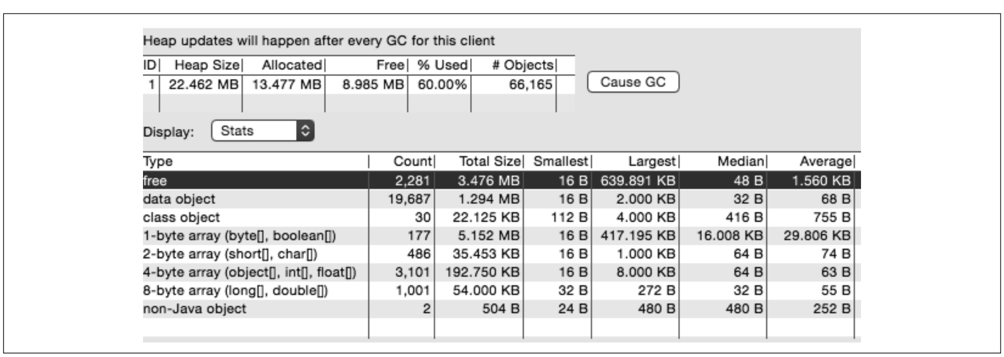


图 5-5：未优化 App 的 Heap Dump 结果

Heap 工具在左侧列出了不同的设备，在右侧展示了一张表格，分析了内存是如何分配的。在表格的下方是一幅条形图，根据大小显示了对象的数量。可以通过研究这张表格来了解内存在 App 中是如何分配的（见图 5-6）。



Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	22.462 MB	13.477 MB	8.985 MB	60.00%	66,165

Cause GC

Display: Stats

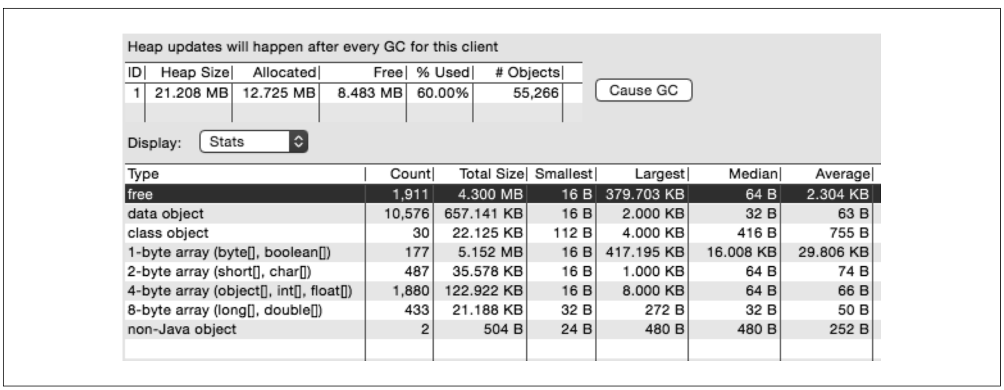
Type	Count	Total Size	Smallest	Largest	Median	Average
free	2,281	3.476 MB	16 B	639.891 KB	48 B	1.560 KB
data object	19,687	1.294 MB	16 B	2.000 KB	32 B	68 B
class object	30	22.125 KB	112 B	4.000 KB	416 B	755 B
1-byte array (byte[], boolean[])	177	5.152 MB	16 B	417.195 KB	16.008 KB	29.806 KB
2-byte array (short[], char[])	486	35.453 KB	16 B	1.000 KB	64 B	74 B
4-byte array (object[], int[], float[])	3,101	192.750 KB	16 B	8.000 KB	64 B	63 B
8-byte array (long[], double[])	1,001	54.000 KB	32 B	272 B	32 B	55 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

图 5-6：Heap Dump 表格：未优化的 App

这些是“Is it a goat?”App 未优化的结果，臃肿的布局增加了额外的对象，而且没有刷新主视图（在设置菜单中所有选项都是有效的）。仅屏幕一项就创建了 22.5MB 的堆。13.7MB 的内存分配给了 66 165 个对象。在下方更大的表格中，可以看到对象、类、数组分别用了多少内存。注意，图片将存储为字节数组的形式，并且 `byte[]` 被分配了最多的内存。

另外一个有趣的特性是，此 App 会保持堆容量的 40% 作为空闲内存，同时已分配内存的大部分（接近 3.5MB）也会作为空闲内存。如果仔细看高亮“空闲”的这一行，你会发现这些分配的空闲空间相当分散。在 2281 个空闲空间中，最小的是 16B，最大的是 639KB。而中位数是 48B，这意味着有一半（或者 1140）的分配空闲空间是超过 48B 的。当新的对象被创建时，只有最小的新对象才能匹配这些空间。所以在 App 运行时，知道内存分配给了哪些对象是非常重要的。

在第 4 章中，“Is it a goat?”App 优化了各种各样的视图文件。优化“More Optimized Layout：RL”视图，同时选择“重绘主视图”选项并取消“创建额外的对象”选项之后，再重新运行 Heap Dump，将会移除大量对象和内存。有多少呢？对比一下图 5-6 和图 5-7 就知道了。



Heap updates will happen after every GC for this client

ID	Heap Size	Allocated	Free	% Used	# Objects
1	21.208 MB	12.725 MB	8.483 MB	60.00%	55,266

Cause GC

Display: Stats

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,911	4.300 MB	16 B	379.703 KB	64 B	2.304 KB
data object	10,576	657.141 KB	16 B	2.000 KB	32 B	63 B
class object	30	22.125 KB	112 B	4.000 KB	416 B	755 B
1-byte array (byte[], boolean[])	177	5.152 MB	16 B	417.195 KB	16.008 KB	29.806 KB
2-byte array (short[], char[])	487	35.578 KB	16 B	1.000 KB	64 B	74 B
4-byte array (object[], int[], float[])	1,880	122.922 KB	16 B	8.000 KB	64 B	66 B
8-byte array (long[], double[])	433	21.188 KB	32 B	272 B	32 B	50 B
non-Java object	2	504 B	24 B	480 B	480 B	252 B

图 5-7：Heap Dump 表格：优化过的 App

App 中的变化有：视图层次和过度绘制大大缩减。对象在运行时创建，而不是在代码中。同时，保证了对无效视图进行更快的垃圾回收处理。

观察全部堆的大小发现，它减少了 1254KB（或者说 5.5%），确保了其在低端设备上的性能。对象数量大概在 11 000 左右或者更低，大部分是数据对象，同时包括大量的 4B 和 8B 数组。在 5.125MB 上的 1B 数组没有变化。图片存储在这 1B 的数组上，所以每 12 张缩略图都存储在堆的这个位置。在两个视图上使用同一张图片，图片内存大小并没有变化，因为每张图片只需要在内存中分配一次（尽管它们在视图层次上被使用多次）。

Heap Dump 工具根据类型将内存的使用分类，如果想要查找内存问题，有时需要一直分离对象直到找到内存问题。而 Allocation Tracker 工具将帮助处理这一类型的问题。

5.3.2 Allocation Tracker

想要在程序运行时发现 App 分配了什么对象，DDMS 中的 Allocation Tracker（分配追踪器）是一个不错的选择。Allocation Tracker 可以在一段时间内追踪每一个被分配内存的对象。这是一个很好的方法，可以用来检查是否有必要创建新的对象，因为创建新的对象有可能导致填满内存或阻塞渲染。

按下 Start Tracking 按钮可以收集分配的列表。执行测试，然后点击 Get Allocations。在一段时间内创建的对象列表和分配的内存将显示在图表中。Allocation Tracker 追踪测试是累计的，所以如果第二次点击 Get Allocations 按钮并且没有先选择 Stop Tracking 按钮的话，先前的结果会添加到第二次的测试中。因此，建议每次测试时重新启动这个工具。图 5-8 是内存分配列表的一个例子。

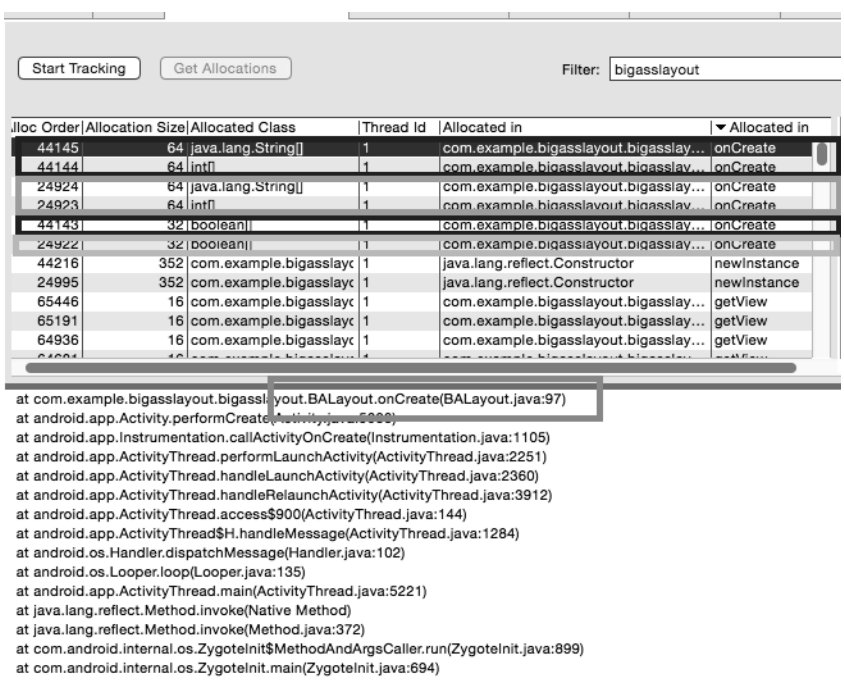


图 5-8：Allocation Tracker 显示的冗余创建的数组（另见彩插）

在图 5-8 的测试中，我运行了“Is it a goat?”App，并且收集了所有从竖屏到横屏再到竖屏的信息。上图的表格可以按照每一列进行排序，并且可以过滤。因为主要的 activity 叫作

com.bigasslayout（回忆一下隐藏着几个大驴子图片的视图层级），所以我将 activity 的名字进行过滤。仔细研究结果（通过对表格的列进行大量排序而找到的规律），发现每次旋转屏幕，都会创建 3 个数组（`string[]`、`int[]` 以及 `byte[]`）。这些数组建立了这些视图，而且并没有发生改变，所以应该将它们保存为静态类型或者存储在配置文件里，以防止它们重复创建。较大的 44KB 的数组（红框内）出现的原因是，在竖屏视图显示的数据比横屏（绿框内，每个数组的大小大概 24 000B）更多。选择表格的一行（如上图，选择表格最顶部的 String 数组），将会在屏幕的底部显示更为详细的信息。在这个示例中，它显示了 BALayout 代码中的第 97 行产生的这个数组（橙色框内）。

这 3 个数组的数据量并不是很大，却是一个简单的例子，可以用来说明如何创建不必要的对象以增加内存的需求量（增加额外的垃圾回收），以及移除它们是如何减少 App 的内存使用量的。在“Is it a goat?”App 中，可以通过在设置菜单中选择“Create Objects During Render”的复选框来复制报告。这样将从保存配置当中移除这些数组，强制 App 每次旋转设备时重新创建这些菜单。取消选择这个复选框将允许保存状态，在每次旋转屏幕的时候，将不会看到这 3 个文件被重新创建。

5.3.3　增加一处内存泄露

我在“Is it a goat?”App 中添加了一个选项，这将会增加一处内存泄露。如下显示：

```
//snip

class Iceberg{
    static ArrayList<byte[]> iceSheet = new ArrayList<byte[]>();
    void sink(){
        byte[] mostlyUnderwater;
        mostlyUnderwater = new byte[2048 * 1024];
        iceSheet.add(mostlyUnderwater);//icesheet每次旋转应增加2MB
        Log.i("iceberg", "Captain, I think we might have hit something.");
    }
}
class CancelTheWatch{
    static Iceberg iceberg;
}
//snip
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
                //snip
        if (memoryLeakTF) {
            //调用内存泄漏类
                        // 当泰坦尼克号取消了时钟，
                        // 它撞到了一个冰山……
            CancelTheWatch NoNeed = new CancelTheWatch();
            Iceberg theBigOne = new Iceberg();
            NoNeed.iceberg = theBigOne;
                        //这会泄漏内存
```

```
                    <snip>
                    //下一行会快速耗尽内存
        NoNeed.iceberg.sink();
    }
```

上文发生了两件事情。通过从 Iceberg 类中调用 theBigOne，然后将 theBigOne 引入 CancelTheWatch 中的静态 Iceberg 内，静态类存活的时间比视图存活的时间长，所以，当屏幕旋转时，视图不能在新视图创建的时候销毁，导致了内存泄露。

Iceberg 泄露并不是很严重。为了从根本上提高 App 的内存堆，发现内存溢出的错误，Iceberg.sink 对象创建了 2MB 字节（mostlyUnderwater）的数组并将其加入到 iceSheet 的 ArrayList 中。在低内存设备中（下面的例子是具有 Jelly Bean 版本的 Samsung Galaxy Note Ⅱ），它很快就会导致崩溃。

```
02-03 02:10:27.650    9399-9399/<app name> D/AbsListView:
                               Get MotionRecognitionManager
02-03 02:10:31.680    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 782K,
        7% free 17078K/18311K, paused 36ms, total 38ms
02-03 02:10:31.680    9399-9399/<app name> I/dalvikvm-heap: Grow heap (frag case)
        to 19.108MB for 2097168-byte allocation
02-03 02:10:31.695    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:10:31.710    9399-9402/<app name> D/dalvikvm: GC_CONCURRENT freed 611K,
        10% free 18514K/20423K, paused 11ms+2ms, total 27ms
02-03 02:10:31.710    9399-9399/<app name> D/dalvikvm:
                               WAIT_FOR_CONCURRENT_GC blocked 11ms
02-03 02:10:31.725    9399-9399/<app name> D/AbsListView:
                                Get MotionRecognitionManager
02-03 02:10:35.440    9399-9399/<app name> D/dalvikvm:
                                GC_FOR_ALLOC freed 39K, 7% free
        19151K/20423K, paused 18ms, total 18ms
02-03 02:10:35.445    9399-9399/<app name> I/dalvikvm-heap:
        Grow heap (frag case) to 21.132MB for 2097168-byte allocation
02-03 02:10:35.470    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:10:35.470    9399-9410/<app name> D/dalvikvm: GC_FOR_ALLOC freed 7K,
        6% free 21191K/22535K, paused 24ms, total 24ms<
```

上面的日志信息显示，当屏幕旋转两次时，App 的内存发生变化。每次旋转后，内存增加了 2MB（从 19MB 到 21MB），堆也增加了 2MB（在 02:10:31.680 以及 2:10:35.445 中可以看出）。在堆增长的前后进行了四次的垃圾回收。GC_FOR_ALLOC 发生表示释放内存，以腾出空间满足内存分配请求。因为堆增长每次会暂停系统 36 毫秒、18 毫秒或者 24 毫秒，所以它们会导致 App 发生卡顿。GC_CONCURRENT 是指一般的垃圾回收，它会周期性地清理对象，其 11 毫秒的暂停足以引起卡顿问题。

继续旋转屏幕，内存也继续膨胀（你可以看到现在已经达到 58.5MB 了），垃圾回收器正在尽一切可能阻止内存溢出的错误：

```
02-03 02:11:23.125    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 659K,
        7% free 57413K/61639K, paused 28ms, total 29ms
02-03 02:11:23.130    9399-9399/<app name> I/dalvikvm-heap:
        Grow heap (frag case) to 58.498MB for 2097168-byte allocation
02-03 02:11:23.145    9399-9399/<app name> I/iceberg:
        Captain, I think we might have hit something.
02-03 02:11:23.160    9399-9402/<app name> D/dalvikvm: GC_CONCURRENT freed 259K,
        8% free 59202K/63751K, paused 12ms+2ms, total 27ms
02-03 02:11:23.160    9399-9399/<app name> D/dalvikvm:
                                WAIT_FOR_CONCURRENT_GC blocked 14ms
02-03 02:11:23.175    9399-9399/<app name> D/AbsListView:
                                Get MotionRecognitionManager

02-03 02:11:28.480    9399-9399/<app name> D/dalvikvm: GC_FOR_ALLOC freed 36K,
        7% free 59705K/63751K, paused 16ms, total 16ms
02-03 02:11:28.480    9399-9399/<app name> I/dalvikvm-heap: Forcing collection of
        SoftReferences for 2097168-byte allocation
02-03 02:11:28.505    9399-9399/<app name> D/dalvikvm: GC_BEFORE_OOM freed 80K,
        % free 59624K/63751K, paused 26ms, total 26ms
02-03 02:11:28.505    9399-9399/<app name> E/dalvikvm-heap:
        Out of memory on a 2097168-byte allocation.
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
                                "main" prio=5 tid=1 RUNNABLE
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | group="main" sCount=0 dsCount=0 obj=0x418b9508 self=0x418a03f0
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | sysTid=9399 nice=0 sched=0/0 cgrp=apps handle=1074749232
02-03 02:11:28.505    9399-9399/<app name> I/dalvikvm:
        | schedstat=( 6822358884 1174852496 11100 ) utm=615 stm=67 core=3


02-03 02:11:28.505    9399-9399/<app name> D/AndroidRuntime: Shutting down VM
02-03 02:11:28.505    9399-9399/<app name> W/dalvikvm: threadid=1:
        thread exiting with uncaught exception (group=0x418b82a0)
02-03 02:11:28.510    9399-9399/<app name> E/AndroidRuntime: FATAL EXCEPTION: main
    java.lang.OutOfMemoryError
            at <app name>.BALayout$Iceberg.sink(BALayout.java:77)
            at <app name>.BALayout.onCreate(BALayout.java:234)
            at android.app.Activity.performCreate(Activity.java:5206)
```

在上文的日志摘要中，App 的内存使用量激增超过了 58MB，随后设备出现了内存溢出。让我们来看看 Android 为防止 App 出现内存溢出的崩溃都做了什么事情。这个 App 试图再分配 2 097 168B 的数组，但是已经没有空间了。首先，Dalvik 强制回收 `SoftReferences`，随后给出了 `GC_BEFORE_OOM` 的警告（这是在内存溢出错误之前最后一次进行垃圾回收的机会），由于垃圾回收器不能为字节数组找到一个可用的 2MB 的内存块，App 就崩溃了。

通常来说，只看日志的话内存泄露不容易被发现，但是有一些专业的工具可用来帮助诊断、找到并解决内存泄露问题。接下来，让我们看看 LeakCanary 和 MAT 工具是如何识别内存泄露的。

5.3.4　更加深层次的堆解析：MAT和LeakCanary

为了诊断 App 在哪里出现了内存泄露，需要分析 App 内存中的所有文件。如果能找出需要释放的文件或者内存中重复的文件，那么就可以在代码中解决这些问题。这样就确保了对象能被正确地释放，或者说确保了内存中文件的复用（而不是内存中存储多个重复的实例）。

为了解析 App 内存中的文件，需要在电脑中保存一个内存堆转储。在监控器（装着一半绿色 Android 液体的圆柱体）中紧邻着 Heap Dump 的图标与之类似，但有一个向下的红色箭头。该图标可以为电脑保存堆转储信息以便进行进一步的解析。

> 已保存的堆转储是 Android 特有的格式文件。要用其他工具打开文件，必须要先进行文件转换。转换工具 hprof-conv 存储于 Android SDK 工具目录下的：
>
> ```
> hprof-conv _<existing_filename> <converted_filename>_
> ```
>
> 如果你是从 Android Studio 的 DDMS 中收集的堆转储，那么就不需要专门转换了，因为在 Android Studio 的 DDMS 下它是自动转换运行的。

当创建堆转储的时候，试着复现严重的内存问题。如果可以控制 App 的大小，或者模仿记录的任何行为，内存数据将会以 hprof 文件格式转存。找出泄露可能会非常棘手，而且需要一直盯着工具，所以说泄露越大，反而越容易被找到。

为了解析堆转储，可以利用 Eclipse 的内存分析工具（MAT）。在 2015 年年初，Square 公司发布了 LeakCanary——一个使得 MAT 的解析更加自动化的开源库。当调试时，该工具就会记录 App 的内存泄露问题。首先，我们要了解如何运用 MAT 发现内存泄露，之后弄清 LeakCanary 是如何简化进程的。

5.3.5　Eclipse内存分析工具——MAT

Eclipse 的内存分析工具（MAT）就像它的名字一样：对内存堆进行详细分析的工具。MAT 是 Eclipse IDE 的一部分，但是如果 Android 的开发迁移到了 Android Studio 上，你可以从 Eclipse.org（https://eclipse.org/mat/）上面下载一个独立的 MAT App 使用。

在 MAT 中打开 hprof 文件时，它的确对文件做了一些处理，并询问你是否需要一个自定义的报告。如果正在查询内存泄露，一般我会选择 Leak Suspects 的报告。它会显示使用内存最多的对象。一旦这些运行起来，打开的工具上方就会出现很多标签页。

MAT 工具在不同的窗口上提供了大量的数据。图 5-9 展示的是主视图上的 Overview 的标签页。它显示的是内存主要消耗的饼图。饼图的每个区域代表一块被分配的内存，点击每

个区域将会看到这块区域的详细信息。最大的内存块是灰色的，代表的是空闲内存。第二大的内存块是 `Iceberg` 类（包含 2 个字节数组的 `ArrayList`），大概占用 4MB 的内存。

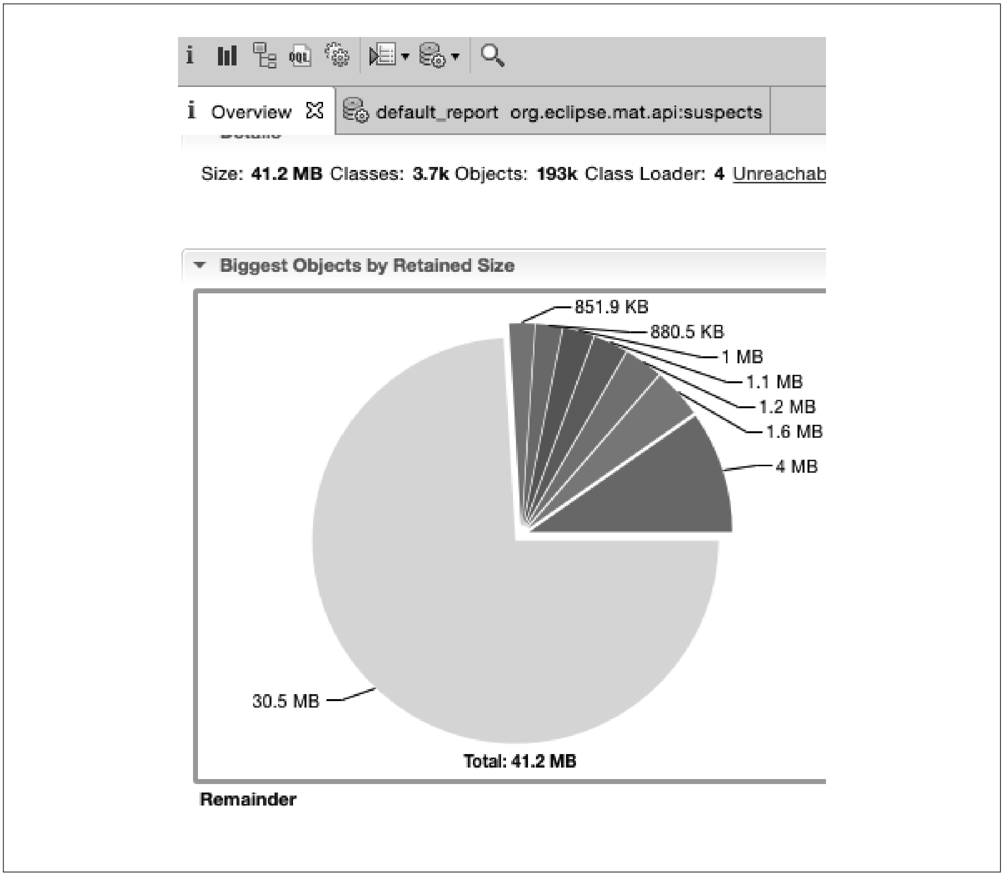


图 5-9：MAT 概况

正如在饼图中 `Iceberg` 类呈高亮状态，Inspector 窗口（见图 5-10）也提供了更多关于 `Iceberg` 类当前引用对象的信息。就像在代码中看到的一样，窗口同样展示了 `iceSheet ArrayList`。

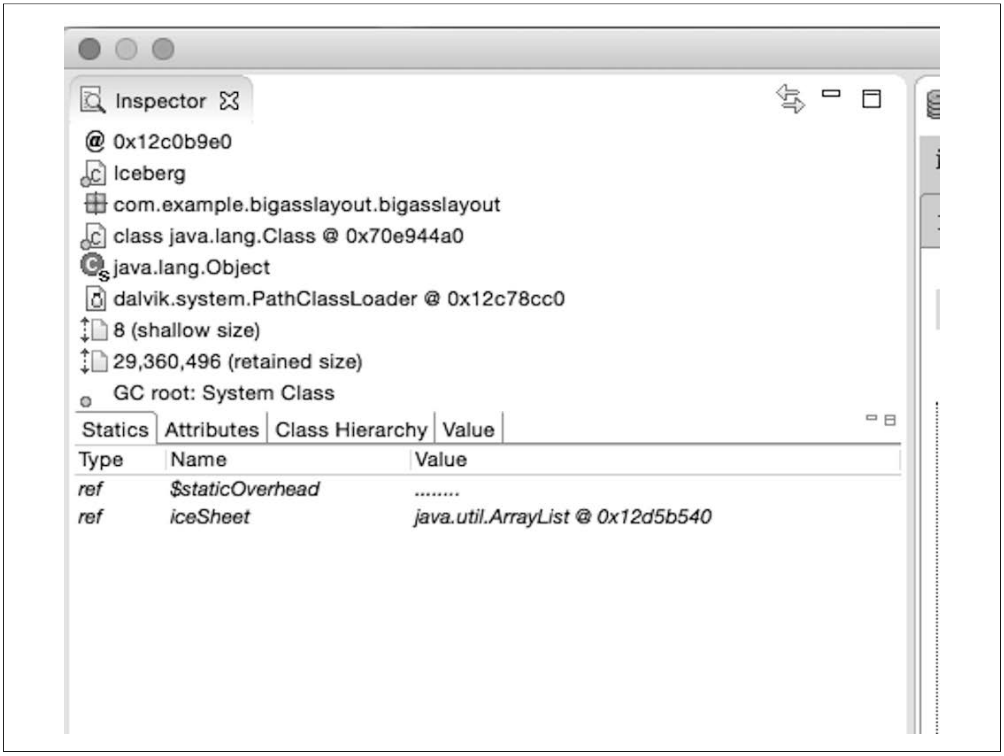


图 5-10：MAT 的 Inspector 窗口

将主视图从 Overview 切换到 Leak Suspect 报告，产生了另一个列出内存泄露猜想（基于使用的内存）的饼图。图 5-11 显示了两个不同堆转储的饼图。左边的饼图是屏幕旋转两次之后的成果，有两个泄露预测，最大的部分大概占用了 27MB（字节数组），Java 类大概占用 6.1MB（Java 类）。右边的图表是屏幕旋转多次之后的结果，字节数组占用的内存大概是 27MB，但是 Java 类的内存分配激增到了 36MB。如果还不知道内存泄露的位置，这看起来是一个很好的发现机会。

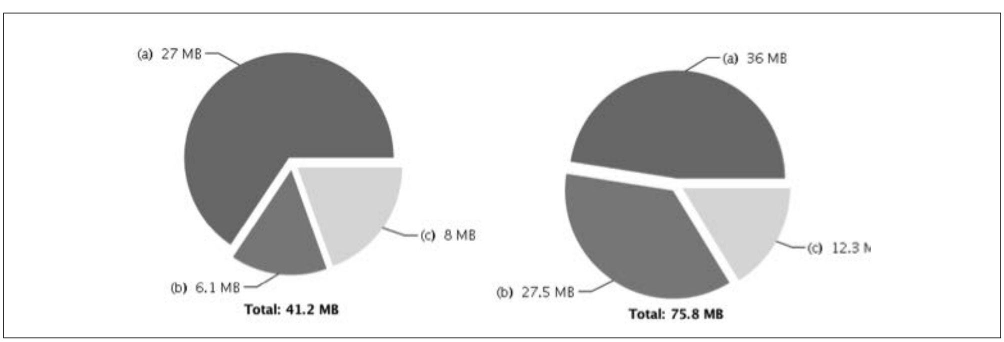


图 5-11：两个内存堆转储的 MAT 泄露猜想图

饼图的下方是一个描述了所有可疑信息（见图 5-12）的黄框。在这个图中，我们将会根据第二次追踪（多次屏幕旋转）继续分析。

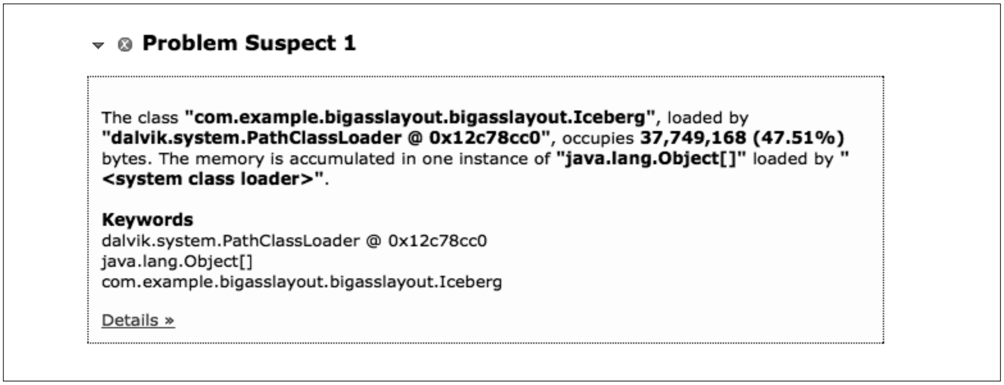


图 5-12：MAT 泄露猜想 1

猜想 1 是 `Iceberg` 类，在一个 Java 对象中使用了 37MB（全部内存的 47%）。点击详细信息的链接可以更进一步地研究猜想的内容。

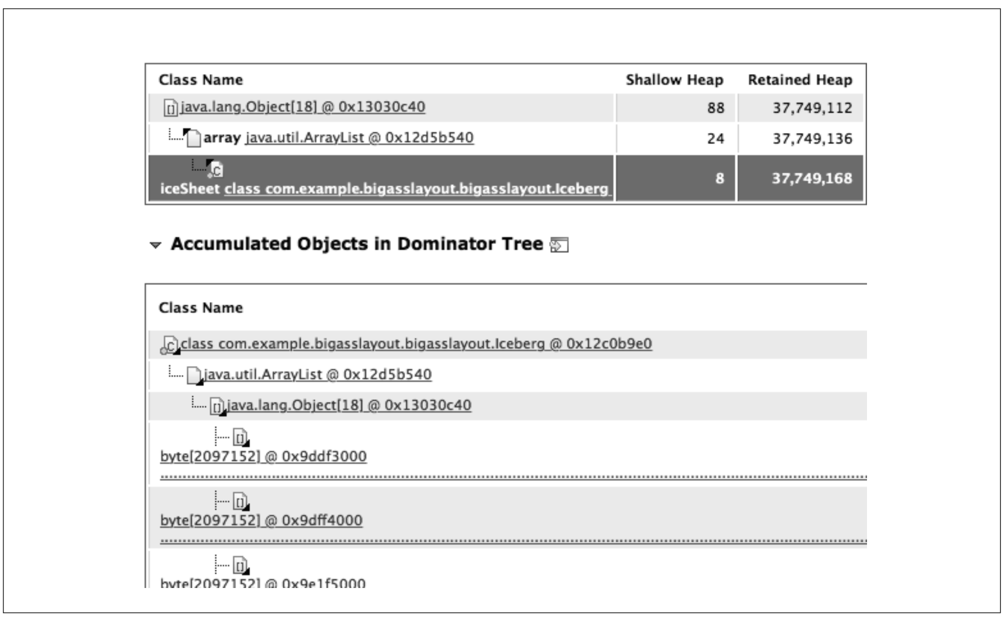


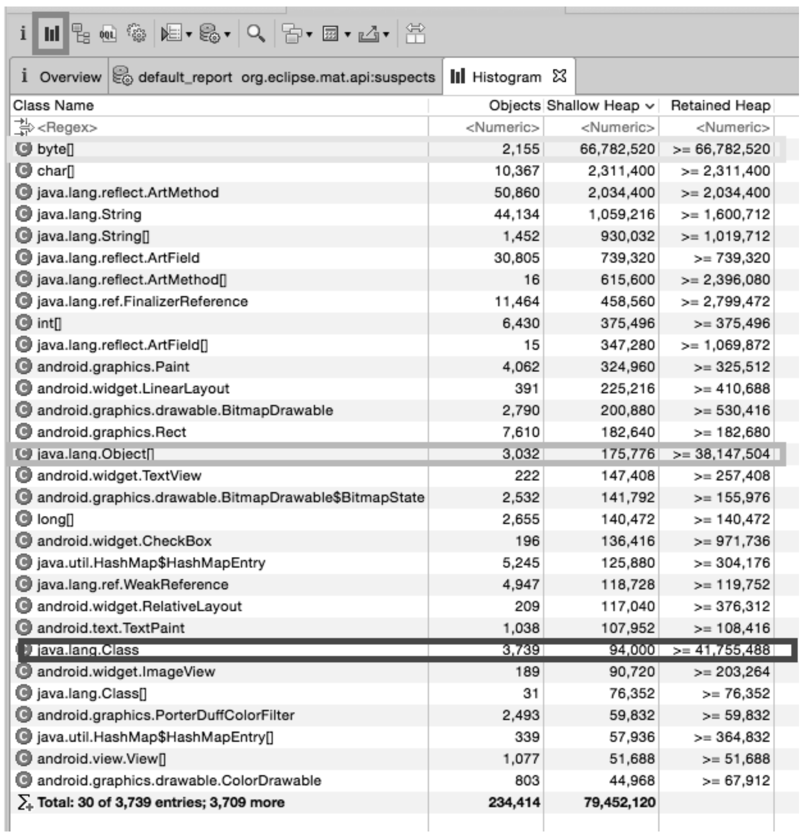
图 5-13：MAT 的泄露视图

在这种情况下，泄露猜测报告确定了问题所在。在 The Shortest Path to Accumulation Point（在内存中对象引用的路径都保持如此）的视图直指 `ArrayList iceSheet` 。当然，在这个例子中，路径并不复杂，但它确实有作用。

当然也有一些巧妙的内存信息：`iceSheet` 有一个 8B 的浅堆，同时有一个 37MB 的保留堆。浅堆是对象所用的内存，而保留堆是对象和所有对象引用对象的内存（在上面的例子中，是 18 个 2.09MB 字节的数组）。就像是树根在土壤中紧紧地抓住了树干部分，依旧在内存中的对象抓住了它们在内存中引用的所有其他对象。这很明显就是内存泄露。

很少有这么简单的例子。如果泄露不是非常明显，则需要更进一步的分析。让我们一起看看 MAT 中可以帮助隔离内存泄露的其他选项。

按下那个长得像条形图表（图 5-14 中，深色方框标记的地方）的图标，一个内存分布图就会被创建。



Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
byte[]	2,155	66,782,520	>= 66,782,520
char[]	10,367	2,311,400	>= 2,311,400
java.lang.reflect.ArtMethod	50,860	2,034,400	>= 2,034,400
java.lang.String	44,134	1,059,216	>= 1,600,712
java.lang.String[]	1,452	930,032	>= 1,019,712
java.lang.reflect.ArtField	30,805	739,320	>= 739,320
java.lang.reflect.ArtMethod[]	16	615,600	>= 2,396,080
java.lang.ref.FinalizerReference	11,464	458,560	>= 2,799,472
int[]	6,430	375,496	>= 375,496
java.lang.reflect.ArtField[]	15	347,280	>= 1,069,872
android.graphics.Paint	4,062	324,960	>= 325,512
android.widget.LinearLayout	391	225,216	>= 410,688
android.graphics.drawable.BitmapDrawable	2,790	200,880	>= 530,416
android.graphics.Rect	7,610	182,640	>= 182,680
java.lang.Object[]	3,032	175,776	>= 38,147,504
android.widget.TextView	222	147,408	>= 257,408
android.graphics.drawable.BitmapDrawable$BitmapState	2,532	141,792	>= 155,976
long[]	2,655	140,472	>= 140,472
android.widget.CheckBox	196	136,416	>= 971,736
java.util.HashMap$HashMapEntry	5,245	125,880	>= 304,176
java.lang.ref.WeakReference	4,947	118,728	>= 119,752
android.widget.RelativeLayout	209	117,040	>= 376,312
android.text.TextPaint	1,038	107,952	>= 108,416
java.lang.Class	3,739	94,000	>= 41,755,488
android.widget.ImageView	189	90,720	>= 203,264
java.lang.Class[]	31	76,352	>= 76,352
android.graphics.PorterDuffColorFilter	2,493	59,832	>= 59,832
java.util.HashMap$HashMapEntry[]	339	57,936	>= 364,832
android.view.View[]	1,077	51,688	>= 51,688
android.graphics.drawable.ColorDrawable	803	44,968	>= 67,912
Total: 30 of 3,739 entries; 3,709 more	**234,414**	**79,452,120**	

图 5-14：MAT 分布图

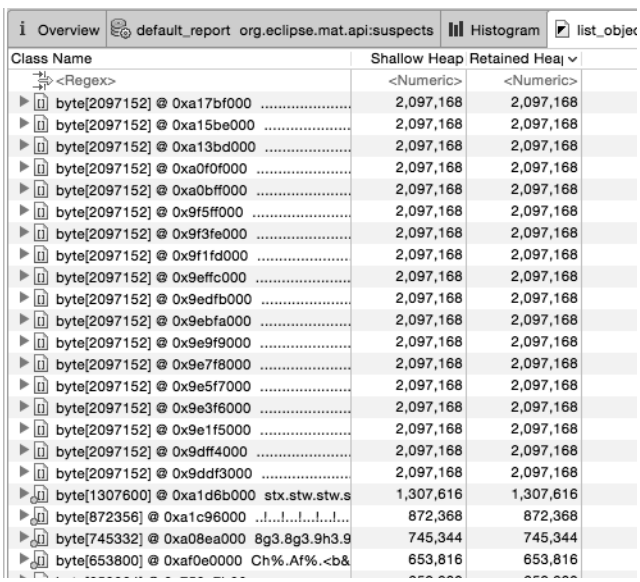
这篇报告根据类分析了内存使用（再次以浅堆和保留堆进行区分）。在图 5-14 中，有几个线索可以用于分析内存泄露。

- byte[]（第一个方框部分）包括了所有的图片（以及在 iceSheet 数组列表中的所有项）。66MB 比预期的要多，这是因为在图 5-11 中，只有 27MB 字节数组作为图片。
- java.lang.Object[]（第二个方框部分）有一个很小的浅堆，但是有大于 38MB 的保留堆。
- java.lang.Class（第三个方框部分）有一个类似的小浅堆，但是却有一个更大的保留堆。

这些线索都表示了小文件，但这些小文件却又对其他对象有庞大的引用。所以我们应该更进一步研究这些类。

如果在直方图上找不到你的 activity（或者感兴趣的类），点击顶部栏的 Regex，然后就可以输入正则表达式来搜索类名。

为了更仔细地检查 byte[] 的对象列表，右击这一行，并且选择 List Objects，然后选择“With Incoming references”。这样将会产生一个根据保留堆排序的新表格（见图 5-15）。



Class Name	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>
byte[2097152] @ 0xa17bf000	2,097,168	2,097,168
byte[2097152] @ 0xa15be000	2,097,168	2,097,168
byte[2097152] @ 0xa13bd000	2,097,168	2,097,168
byte[2097152] @ 0xa0f0f000	2,097,168	2,097,168
byte[2097152] @ 0xa0bff000	2,097,168	2,097,168
byte[2097152] @ 0x9f5ff000	2,097,168	2,097,168
byte[2097152] @ 0x9f3fe000	2,097,168	2,097,168
byte[2097152] @ 0x9f1fd000	2,097,168	2,097,168
byte[2097152] @ 0x9effc000	2,097,168	2,097,168
byte[2097152] @ 0x9edfb000	2,097,168	2,097,168
byte[2097152] @ 0x9ebfa000	2,097,168	2,097,168
byte[2097152] @ 0x9e9f9000	2,097,168	2,097,168
byte[2097152] @ 0x9e7f8000	2,097,168	2,097,168
byte[2097152] @ 0x9e5f7000	2,097,168	2,097,168
byte[2097152] @ 0x9e3f6000	2,097,168	2,097,168
byte[2097152] @ 0x9e1f5000	2,097,168	2,097,168
byte[2097152] @ 0x9dff4000	2,097,168	2,097,168
byte[2097152] @ 0x9ddf3000	2,097,168	2,097,168
byte[1307600] @ 0xa1d6b000 stx.stw.stw.s	1,307,616	1,307,616
byte[872356] @ 0xa1c96000 ..!...!...!...!...	872,368	872,368
byte[745332] @ 0xa08ea000 8g3.8g3.9h3.9	745,344	745,344
byte[653800] @ 0xaf0e0000 Ch%.Af%.<b&	653,816	653,816

图 5-15：MAT 列出的 byte[] 的对象

在列表的顶部，可以看到由屏幕旋转得到的 18 个 2MB 的数组。如果想在垃圾回收的时候找到阻塞的根对象，对一个对象右击，并且选择“Path to GC Roots”→“excluding weak references”（在垃圾回收的过程中，弱引用不会阻塞对象）。这将会打开一个新的窗口，如图 5-16 所示。

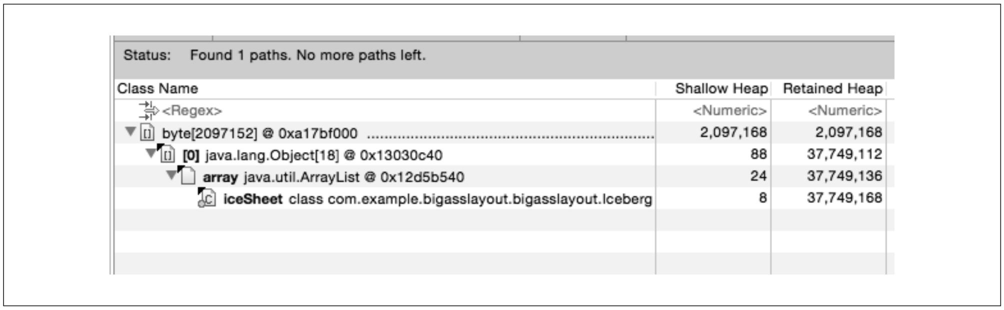


图 5-16：垃圾回收根部 2 字节数组

垃圾回收根部路径再次确认了 `iceSheet` 就是内存泄露的罪魁祸首。上图选择研究第一个字节数组，报告的第二行显示它在含有 18 个元素的数组列表的第 0 个位置。最后一行又出现了 `iceSheet` 的名字。我们再一次找到了内存泄露的原因。hprof 文件保存在本书的 GitHub 仓库里（https://github.com/dougsillars/HighPerformanceAndroidApps）。我将追踪 `iceSheet` 内存泄露的 `java.lang.Object` 和 `java.lang.classes` 留作一个练习，大家可以根据与上面相同的步骤找到同样的答案。

> 如果你认为字节数组中的泄露和某个图片相关（因为所有的图片都以字节数组的形式存储在内存中），但是你不知道是什么图片引起了问题，下面有方法可以将字节数组转化为图片。这个字节数组有一个 `android.graphics.bitmap` 类型的“mbuffer”内部对象。点击这个对象将会在 Inspector 视图中显示这个对象的宽和高。然后右击这个字节数组选择 Copy → “Save value to file”（以扩展名 .data+ 的形式保存）。你可以用类似 GIMP 的工具打开这个文件，输入宽和高的值，GIMP 将会显示出字节数组中隐藏的图片。

要想学习 Android 是如何处理内存分配，并找到方法优化 App 的内存分配，使用 Eclipse MAT 是一个不错的方法。但是在版本迭代快速的时代，你可能没有时间学习和调研一种新的工具来诊断内存泄露。好在 Square 团队开源了 LeakCanary，一款使 MAT 做的大部分事情能自动化输出的测试工具。

5.3.6 LeakCanary

Square 公司开发的 LeakCanary 被用来减少 App 遇到的内存泄露错误。他们在不同的设备上发现了相同的崩溃，然后在 MAT 上做了一些实质性的实验来发掘是什么引发了泄露。这种方法比较慢，而他们想要在发布给最终的用户之前找到这个内存泄露，LeakCanary 就诞生了。对于内存泄露，它就像是“煤矿中的金丝雀”：在内存溢出、崩溃之前，就可以嗅出内存泄露。自从使用了 LeakCanary，Square 公司报告（https://corner.squareup.com/2015/05/leak-canary.html）显示，OOM 崩溃下降了 94%。让我们一起看一下这个工具

是如何运作的。

根据 Square 的说明（https://github.com/square/leakcanary），启动和运行 LeakCanary 非常简单。我在“Is this a goat?”App 上运用了它，并且放到了 GitHub 上面。

在 build.gradle 文件上添加两个依赖：

```
debugCompile 'com.squareup.leakcanary:leakcanary-android:1.3.1'
releaseCompile 'com.squareup.leakcanary:leakcanary-android-no-op:1.3.1'
```

在“Is it a goat?”App 的应用类中，添加如下：

```
//LeakCanary引用监视器
public static RefWatcher getRefWatcher(Context context) {
    AmiAGoat application = (AmiAGoat) context.getApplicationContext();
    return application.refWatcher;
}
private RefWatcher refWatcher;


@Override public void onCreate() {
    super.onCreate();
    //在App创建上 - 打开LeakCanary

    refWatcher = LeakCanary.install(this);
}
```

然后给 `CancelTheWatch` 类和 `Iceberg` 类添加了特定的引用 watcher：

```
//LeakCanary监测变量
RefWatcher wishTheyHadAWatch = AmiAGoat.getRefWatcher(this);
wishTheyHadAWatch.watch(NoNeed);

RefWatcher icebergWatch = AmiAGoat.getRefWatcher(this);
icebergWatch.watch(theBigOne);
```

现在，当我运行“Is it a goat?”App，打开内存泄露，旋转屏幕时，事情发生了。在短暂的延迟之后，LeakCanary 输出堆转储和相应的分析。写在日志上的报告如下：

```
05-25 15:43:28.283  17998-17998/<app>I/iceberg:
                   Captain, I think we might have hit something.
05-25 15:43:51.356  17998-18750/<app> D/LeakCanary: In <app>:1.0:1.
    * <app>.Iceberg has leaked:
    * GC ROOT static <app>.CancelTheWatch.iceberg
    * leaks <app>.Iceberg instance
    * Reference Key: 52614375-1531-47b1-96d7-4ec986861794
    * Device: motorola google Nexus 6 shamu
    * Android Version: 5.1 API: 22 LeakCanary: 1.3.1
    * Durations: watch=5443ms, gc=154ms, heap dump=2864ms, analysis=14302ms
    * Details:
    * Class <app>.CancelTheWatch
```

```
|   static $staticOverhead = byte[] [id=0x12c9f9a1;length=8;size=24]
|   static iceberg = <app>.Iceberg [id=0x1317e860]
* Instance of <app>.Iceberg
|   static $staticOverhead = byte[] [id=0x12c88e21;length=8;size=24]
|   static iceSheet = java.util.ArrayList [id=0x12c267a0]
```

这个跟踪显示了关于设备的一切信息。据此追踪可知，`Iceberg` 类有泄露，整个过程花费的时长（垃圾回收用了 154 毫秒，收集堆转储用了 2 秒，分析用了 14 秒），以及哪些对象在类中引起了泄露。GitHub 文档一步步地引导你向服务器机群上报这些泄露和堆转储。（注意，对于明显的延迟原因，内存泄露应该在调试版本就进行修复，这对内部测试具有重大的意义！）最终，这个报告将会在设备的通知栏进行提示，并以一个叫“Leaks”（见图 5-17）的新 App 在 App 列表中出现。

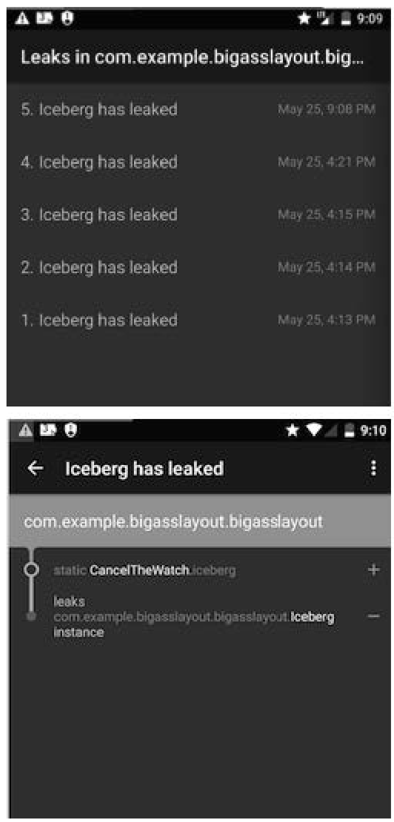


图 5-17：LeakCanary 截图：摘要（顶部）和详情（底部）

LeakCanary 将存储设备最开始的 7 个泄露，并且会有一个菜单和其他人分享这个泄露和堆转储的信息。在内部测试中使用 LeakCanary 将帮助你发现 MAT 发现不了的内存泄露问题，快速定位 App 中的内存泄露，减少崩溃量，并提高 App 的性能。

5.4 小结

直到现在，发现内存泄露问题的唯一方法仍是研究所有内存溢出的崩溃，并且利用 MAT 仔细研究内存引用的相关问题。MAT 依旧是一个优秀的工具。理解 MAT 暴露的内存链接是非常重要的。然而，LeakCanary 可以代替不断使用的 MAT 来测试内存问题。

通过仔细研究 Android App 如何控制内存操作，App 在内存受限的设备上将会更加高效地运行，内存溢出的崩溃也会相应地减少。限制对象并确保它们有适当的生命周期，可以降低垃圾回收对 App UI 的影响，避免垃圾回收阻塞 App 的主线程。最后，使用 Allocation Manager、LeakCanary 以及 MAT 等工具，可以识别出正在泄露内存的对象和类。

第6章

CPU与CPU性能

在前面几章中，我们已经讲到了电池、UI 和内存管理性能，以及这些性能优化如何有助于减少崩溃和提高 App 性能。在这一章中，我们将继续讨论如何构建高性能的 Android App。我们将会介绍 Android 设备的一个重要部分——CPU。CPU 是设备的大脑，它用来处理所有代码，从而构建 App。同时，如何合理地使用 CPU 是优化程序的另一个难点。

事实上，芯片供应商在不断地提升芯片的性能，同时也在努力解决电池续航和芯片发热的问题。近几年来，Android 设备无论是在性能还是效率上，进步都非常迅猛。

过去几年，四核、八核、十核 CPU 在市场上越来越常见。不同于 PC（所有的 CPU 都是相同的，而且可以计算任何内容），ARM 构架的移动芯片用不同的 CPU 计算不同的任务。ARM 称这样的芯片设计为 big.LITTLE，这很形象地形容了它的工作原理。当一个小型后台任务执行时（例如检查邮件），一个低功耗、更高效的 CPU 会被分配执行这个任务。当用户观看视频或者玩游戏时，高性能的核心将被激活。通过将小型任务转交给 LITTLE 处理器，并且只用高性能 CPU 运行耗电任务，可以为设备节省电量。令开发者欣慰的是，所有这些事情都由内核控制，内核会为你选择合适的处理器。

在第 5 章我们看到，在自动管理内存的环境下，我们仍然可以对内存进行优化。同样，我们不能假设 App 代码会正确地利用设备上的 CPU。让 App 恰当地使用 CPU 是十分必要的。在 6.1 节中，我们会关注如何了解 Android 设备上的 CPU 总体占用率和某个单独 App 的 CPU 占用率，以及如何确定 App 中哪个线程或进程正在大量占用 CPU。我们会关注为什么错误使用 CPU 会阻止渲染，引发令人畏惧的“应用无响应”（Application Not Responding，ANR）警告，甚至使 App 崩溃。

6.1 检测CPU占用率

让我们再从较宏观的层面观察，看看你的 App 与内核是如何同其他 App 协作使用 CPU 的。通常，我们使用 Linux 的 `top` 命令查看设备的 CPU 占用率：

```
demo$ adb shell top -n 1 -m 10 -d 1

User 58%, System 14%, IOW 0%, IRQ 0%
User 157 + Nice 6 + Sys 41 + Idle 75 + IOW 1 + IRQ 0 + SIRQ 0 = 280
```

运行 1 次命令（`-n 1`），可以获取 1 秒内（`-d 1`）10 个 CPU 占用率最高的 App（`-m 10`）。我们看到，用户占用了 58% 的 CPU，内核空间占用了 14%。第二行显示了调度器在状态间切换花费的时间（几十毫秒）。最大数值可能是 CPU 数的 100 倍。我们发现活跃的进程一共有 280 个，由于测试运行在 Nexus 6（四核 CPU）上，最大值是 400。

现在，我们看一下占用率前十的 App：

```
  PID PR CPU% S  #THR     VSS     RSS PCY UID      Name
15252  1  32% S    16 1581536K  93324K  fg u0_a109  com.example.isitagoat
 1952  0  20% S    97 1708552K 136668K  fg system   system_server
15987  2   2% R     1   4464K   1108K     shell    top
 2413  2   2% S    32 1650148K  76044K  fg u0_a11   com.google.process.gapps
 3010  1   2% S    41 1810248K 179400K  fg u0_a28
                                   com.google.android.googlequicksearchbox
 3384  1   2% S    47 1621432K  83928K  fg u0_a11   com.google.process.location
 2586  1   2% S    26 1566872K  93088K  fg u0_a91   com.elvison.batterywidget
 2125  0   1% S    32 1698300K 166068K  fg u0_a24   com.android.systemui
  267  1   1% R    15 227172K  17060K  fg system   /system/bin/surfaceflinger
 6256  1   0% S    49 1603916K  83816K  fg u0_a28
                                   com.google.android.googlequicksearchbox
```

如上表所示，“Is it a goat?”App 占用了 32% 的 CPU，系统占用了 20%，`top` 命令占用了 2%，后台和 Google App 也占用了一部分。在 App 运行时执行这个操作是一种快速、粗略的占用率查看方式。我们还发现，这些 App 都有一个相同的策略（PCY）——`fg`，这意味着某种程度上，它们都是前台进程。

这是一个好的开头，如果想要更深入地了解 App 的 CPU 占用率，可以使用 `dumpsys` 命令查看更详细的信息：

```
adb shell dumpsys cpuinfo

adb shell dumpsys cpuinfo
Load: 12.28 / 11.64 / 11.56
CPU usage from 11368ms to 4528ms ago with 99% awake:
  0.3% 1531/mediaserver: 0% user + 0.3% kernel / faults: 1093 minor 1 major
  130% 15754/com.coffeestainstudios.goatsimulator: 111% user + 19% kernel /
     faults: 130 minor
  10% 306/mdss_fb0: 0% user + 10% kernel
  9.8% 267/surfaceflinger: 4.5% user + 5.2% kernel
```

```
  4.5% 1952/system_server: 1.4% user + 3% kernel / faults: 65 minor
  0.8% 19261/kworker/0:1: 0% user + 0.8% kernel
  0.7% 2982/com.android.phone: 0.2% user + 0.4% kernel / faults: 181 minor
  0.5% 158/cfinteractive: 0% user + 0.5% kernel
  0.5% 18754/kworker/u8:4: 0% user + 0.5% kernel
  0.4% 205/boost_sync/0: 0% user + 0.4% kernel
  0.4% 211/ueventd: 0.2% user + 0.1% kernel
  0.4% 2586/com.elvison.batterywidget: 0.2% user + 0.1% kernel /
  faults: 121 minor
<snip>
```

结果第一行分别显示了 CPU 在过去的 1、5、15 分钟的平均负载。之后显示的是在将近 7 秒内所有 App 的 CPU 占用率（由于空间原因被截断了）。你可以看到每个 App 占用 CPU 的百分比（如果不止运行在一个核心上，它可以超过 100%），以及用户态和内核态的比例。

像我们见过的其他大多数命令行接口一样，`cpuinfo` 可以通过开发者选项启用屏幕覆盖层（见图 6-1）。数据基本和命令行是相同的，不同之处是屏幕顶部会显示一些色条，它显示了 CPU 耗费在用户代码（绿色）、内核（红色）、中断（蓝色）的时间。这对确定 IO 阻塞的发生是很有帮助的，你可以在屏幕上及时地看到这些事情的发生。



图 6-1：CPU 信息概况

6.2　使用Systrace分析CPU

`top` 和 `cpuinfo` 可以提供 App 使用内存的基本情况，但是我们仍然需要深入到 CPU 核心，

观察 App 运行时它们在做什么。4.6.3 节中讨论的工具有助于我们发现 UI 中的卡顿 [1]。我们也可以用 Systrace 分析为什么 CPU 可以阻塞渲染并且引起掉帧和卡顿。当我们观察 UI 的时候，这些线条就会隐藏起来。接下来，让我们进一步观察它们。这一章中描述的轨迹都可以在本书的 GitHub 仓库（https://github.com/dougsillars/HighPerformanceAndroidApps）中找到（轨迹 4 没有发生卡顿，轨迹 7 发生了卡顿）。

在所有 Systrace 的顶部（与图 4-22 类似），都可以看到每一个 CPU 核心的使用情况（见图 6-2）。

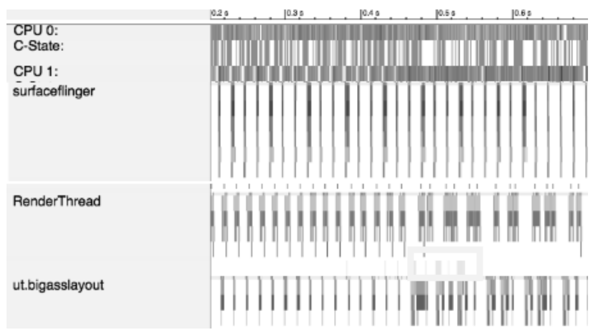


图 6-2：没有卡顿的 Systrace 视图

当我用 Systrace 分析“Is it a goat?”App 中的常规视图时，CPU0 和 CPU1 都显示被占用。但这些大量的计算都很迅速，并且没有阻塞 UI。视图创建的时间非常平均，`SurfaceFlinger` 和我们预想的一样，每 16 毫秒就把视图发送给 GPU。在这两行 CPU 计算中，每一条色带表示一个 App。你可以在底部的菜单中选中它们，或者放大来查看和它们关联的程序名字（见图 6-3）。

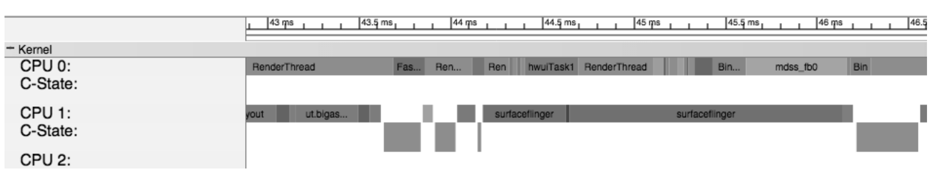


图 6-3：Systrace 的 CPU 视图（另见彩插）

注意，图 6-3 的时间轴总共只有 3.5 毫秒（最大的 tick 是 0.5 毫秒，最小的只有 0.1 毫秒）。我们可以看到 CPU 在这么短的时间内执行的操作（通过颜色）：

注 1：Android 开发团队在 Android 4.1 黄油计划中提到，当 VSYNC 信号发出时，CPU 还没准备好渲染下一帧，这种情况被称为卡顿。——译者注

- 紫色的是“Is it a goat?”App
- 蓝色的是 `RenderThread`
- 砖红色的是 `SurfaceFlinger`
- 还有其他占用时间很短的进程（很多在图中仅表现为一条竖线）

仔细看图 6-3，你会发现 `RenderThread` 和 ut.bigasslayout 的线条在轨迹中间某些地方变得更长了。原因是这时候我正在滑动屏幕改变滚动的方向。

在下一个 Systrace 示例中，我加入了斐波那契数列计算。第 5 次和第 10 次的计算量非常大，以至于造成了滑动时严重掉帧。计算图 6-4 的时长要大于图 6-3，但是渲染的视图却更少。在最初的 100 毫秒内，没有卡顿发生，一切运行正常。但是从大约 150 毫秒开始，UI 因为计算 8 位数的斐波那契数列开始变得卡顿。由于“Is it a goat?”App 忙于处理大量计算，CPU0（和后来的 CPU2）变成了紫红色。App 一直停留在绿色的 obtainView 状态，因为其一直在尝试渲染视图。

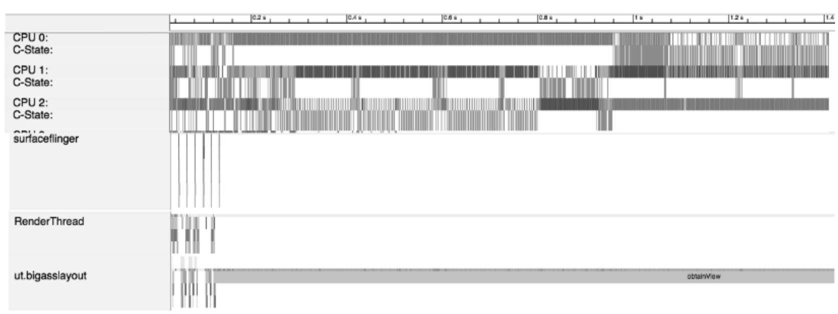


图 6-4：有卡顿的 Systrace 视图（另见彩插）

如果把图 6-4 中的绿色长条 obtainView 放大看，你会发现一条非常细而且颜色不同的线。图 6-5 中，我们看这段 15 毫秒的轨迹，在绿色的 obtainView（最下面的一行）上面，有深绿色和蓝色的指示条。这些指示条表示了 CPU 现在所处的状态。蓝色的片段表明进程处于就绪状态，绿色表明 App 正在使用 CPU。那两条竖线之间代表紫红色程序占用 CPU 的精确时间。Systrace 显示，在 obtainView 阻塞 App 更新界面的时候，有成百的小进程在运行。由此看来，当 UI 被阻塞时，也有可能是因为另一个更复杂的 App 里面的线程阻塞了 UI 渲染。

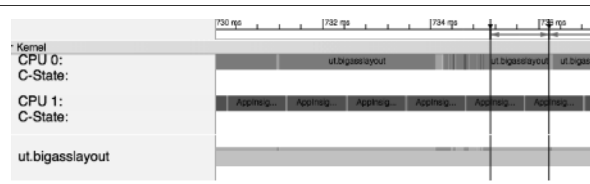


图 6-5：在 Systrace 中的 App 的 CPU 状态图

知道了有一个进程正在阻塞渲染后，我们现在可以使用 Traceview App 更深入地诊断问题。Traceview 有两种模式，它们用不同的方式显示相同的信息。同时讨论两个工具是很有必要的，因为你可能更受用于其中一个。

6.3　Traceview（遗留的监视器DDMS工具）

如果你看过 Android CPU 优化的视频，就应该见过这个工具。它诞生于 Android 初期，直到现在它的实用性仍然是无与伦比的。Android Studio 的用户可以在 SDK 的 Monitor 工具中轻松地找到它。你可以通过选择 App，单击一个由三条水平线和一个白点（还有一个红点）组成的图标（在图 6-6 上方的方框内）来使用它。然后它会弹出一个对话框，请你选择轨迹的类型。第一个选项可以在每 x 毫秒（默认 1000 微秒）对 VM 正在运行的所有进程采样。这是在 CPU 性能有限的设备上的最佳方案，你还可以采集一段更长时间的轨迹。在这个示例中，我们已经选择了第二个选项，把每一个方法的起始和结束都记录下来。但这样会带来更大的额外开销，甚至在性能最强劲的设备上也会增加 App 的延迟（这一节中的轨迹来自于 Nexus 6 5.0.1）。

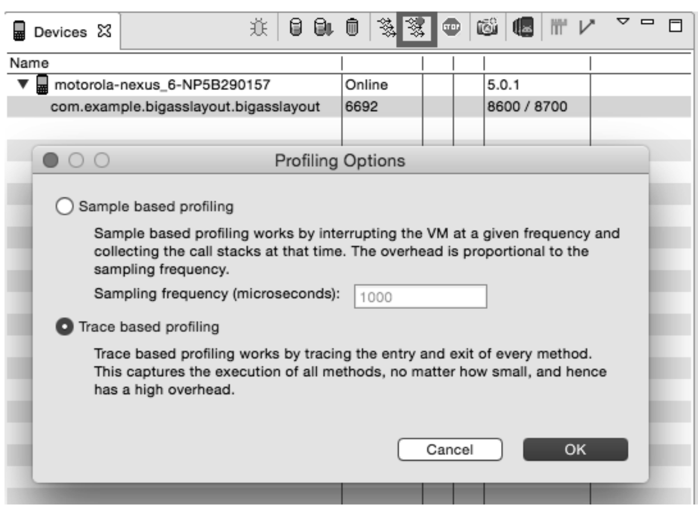


图 6-6：启动 Traceview

打开了 Traceview 之后，运行你要测试的 App，再次点击相同的按钮来停止测试。过一会，DDMS 窗口中间会显示一条轨迹。每个线程都会在顶部单独显示一行（比如“Is it a goat?”App，它只有主线程）。每个方法都用不同的颜色标记了（这一点放到最大的时候可以看到，方法的开始和结束都是黑色线条）。见图 6-7。

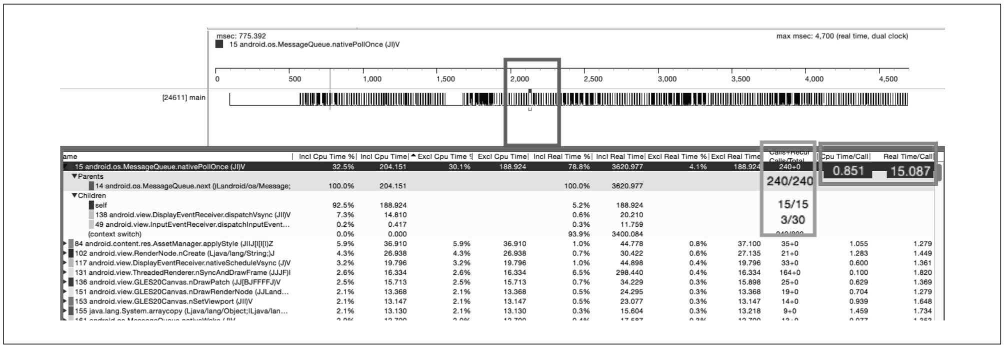


图 6-7：Traceview 概况图

图 6-7 中，Traceview 下方的图表列举了所有方法，你可以看到每个方法分别是由哪个方法调用的，以及它调用了哪个方法。图 6-7 是“Is it a goat?”App 正常操作下的 Traceview 分析图。图中标记了方法 15（`android.os.MessageQueue.nativePollOnce`），它由 `MessageQueue.next` 调用，还调用了分发 `DisplayEvents` 和 `InputEvents` 的方法。下面列出了几种调用方法占用 CPU 的方式。

- Inclusive CPU Time
  花费 CPU 的时间，包括其中调用其他方法的 CPU 时间。
- Exclusive CPU Time
  花费 CPU 的时间，不包括其中调用其他方法的时间。
- Inclusive Real Time
  实际花费的时间，包括其中调用其他方法的时间（对比仅仅花费 CPU 时间）。
- Exclusive Real Time
  实际花费的时间，不包括其中调用其他方法的时间（对比仅仅花费 CPU 时间）。
- Calls + Recursive Calls/Total Calls
  总调用次数。
- CPU Time/Call
  平均每次调用花费的 CPU 时间。
- Real Time/Call
  平均每次调用花费的实际时间。[2]

注 2：CPU Time 和 Real Time 的区别是，CPU Time 是 CPU 执行方法本身代码花费的时间，但在实际过程中，CPU 可能由于某些原因（比如上下文切换、垃圾回收等），消耗额外的时间。Real Time 指的是从方法调用开始，到方法返回的时间。在这个例子中，92.5% 的时间都花费在了上下文切换上。——译者注

“Calls”列显示每个方法被调用了多少次。方法 15 被调用了 240 次。它还调用了方法 138 一共 15 次（并且是方法 138 的唯一调用者）。它调用了方法 49 一共 3 次（其他方法调用了方法 49 共 27 次）。方法 15 本身占用了 CPU 188.924 毫秒，总共 204.151 毫秒（因为调用方法 138 消耗了 14.8 毫秒）。每次调用方法平均花费 CPU 0.851 毫秒，实时 CPU 时间是 15.087 毫秒。

当你选中了任何一行（比如第 15 行），在 Traceview 中会高亮显示该方法的每一次调用。换一种方式，如果你在图中单击方法 15 被调用的地方，也可以看到同样的现象。在图 6-7 中大约 2100 毫秒的地方，可以看到一次深色方框内的调用。Traceview 用在一个空架子上加一个深灰色条的方式来突出方法。因为这个方法是渲染视图的一部分，所以这个方法最好在少于 16 毫秒的时间内完成。我们仔细观察 500 毫秒内，每一帧的绘制过程都调用了该方法（图 6-8 高亮显示了方法 15 的每次调用）。

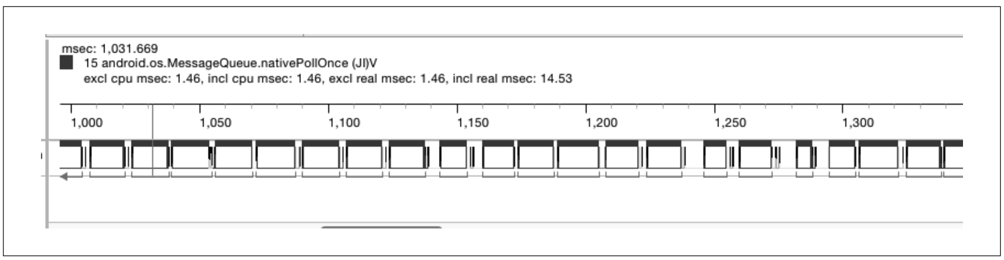


图 6-8：放大图 6-7 的高亮方法，Traceview 显示它每隔 17 毫秒调用一次

现在我们已经知道怎么用 Traceview 分析 App 的性能了，我们看一下在“Is it a goat?”App 打开斐波那契计数器的情况下，Traceview 的性能参数。如图 6-9 所示，两者的不同是显而易见的。

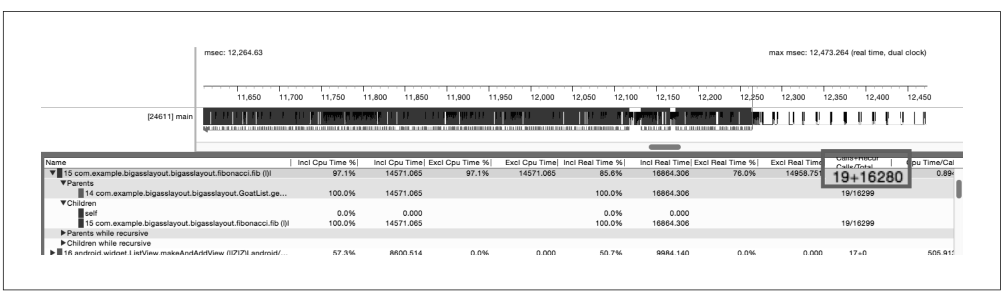


图 6-9：开启斐波那契计算的“Is it a goat?”App 的 Traceview 视图

在 11 600~12 250 秒内，递归的斐波那契数列计算完全占据了主线程，在 Traceview 中的黑线已经变得非常密集。这种情况下，我已经选中了斐波那契进程，每一次调用都在图 6-9 中显示出来。就像我们在 Systrace 中看到的一样，这些调用阻塞了这个 App 的所有其他方法。从 12 250~12 450 秒，又看到了我们愿意看到的场景——主线程上又呈现了有规律的

16 毫秒周期——这意味着没有卡顿发生。

从 Traceview 下方的图表可知，斐波那契方法被调用了 19 次。由于这是一个递归方法，所以它在这个过程中还调用了自身 16 280 次（红色方框内放大的文字）。整个轨迹大概有 19 秒，但是近 17 秒的时间都花费在了这个方法上。如果我们真的需要为数据提供一个斐波那契数，需要用一个更快、CPU 开销更小的方法。

6.4 Traceview（Android Studio）

Android Studio 0.2.10 版本引入了一个新的 Traceview，替换了 6.3 节中讲到的 DDMS/Monitor Traceview。新的 Traceview 使用火焰图（flamecharts）显示轨迹。在 Traceview 的 Monitor 版本中，我们可以很清楚地看到一个方法的直接调用关系，但是间接一级的调用（或者其他更深层次的关联）就很难辨别了，除非非常深入地研究这个图表。火焰图不仅在水平轴上显示了每次方法或进程需要的时间，还在纵轴上显示了所有的调用层次。这使方法之间的相互作用具有更好的可读性。

在 Android Studio 上执行轨迹录制非常容易。与 DDMS 中的红点图标不同，Android Studio 上的图标是一个秒表。点击秒表，Traceview 就开始了。运行轨迹，再单击一次秒表，轨迹就会停止（新版本中没有更改采样率的选项了）。Traceview 会在 Android Studio 的主界面上显示。可以看到，这与之前的 Traceview 截然不同（见图 6-10）。

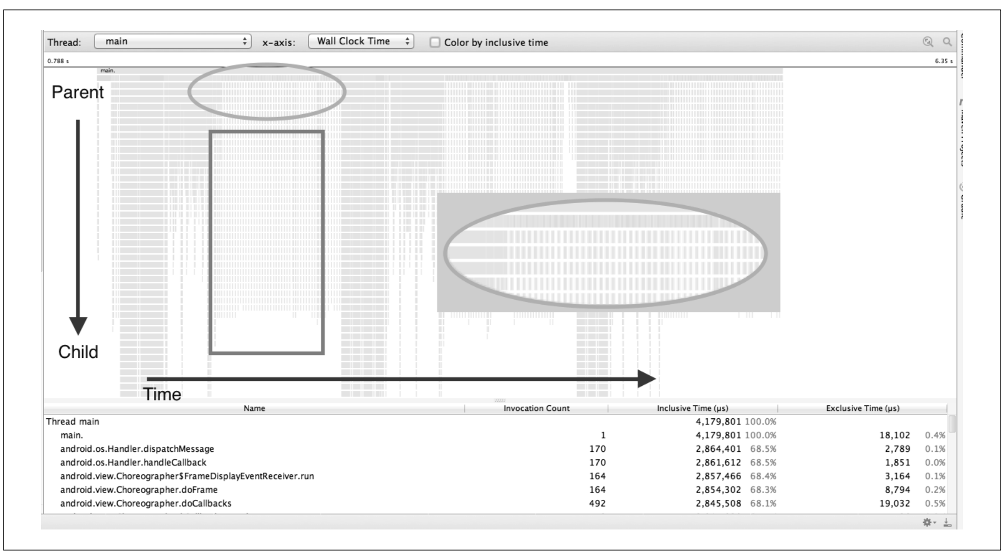


图 6-10：Traceview 火焰概况图

在原始的 Traceview 中，查找方法的直接调用关系已经在图表中做好了。现在，调用者在上方显示，被调用者在下方显示。每个方法的水平条表示调用这个方法花费的时间。每个

线程是分离的（我们可以在顶部的下拉菜单中查看）。线程被标记为红色、橙色、黄色和绿色（依次从慢到快）。这些水平条表示的时间默认为自身方法所花费的时间，在第二行中，有一个高级方法——`MessageQueue.next` 方法。这个方法有大量的调用，因为它负责排队，并等待每个视图绘制完成。图片中放大的椭圆高亮显示了一系列普通方法的根方法，以及大量的依赖（方框中）。这些都是当你滑动到列表底部时，反弹动画的普通调用。放大的区域显示，橙色的 `MessageQueue.next` 在每一帧动画之间执行。

`GoatList` 方法在“Is it a goat?”App 的每一行都有显示。我们可以通过查找方法快速地在火焰图中定位 `GoatList`。在图 6-11 中，蓝色高亮的长条表示 `GoatList` 方法被调用（图中共有 8 处）。

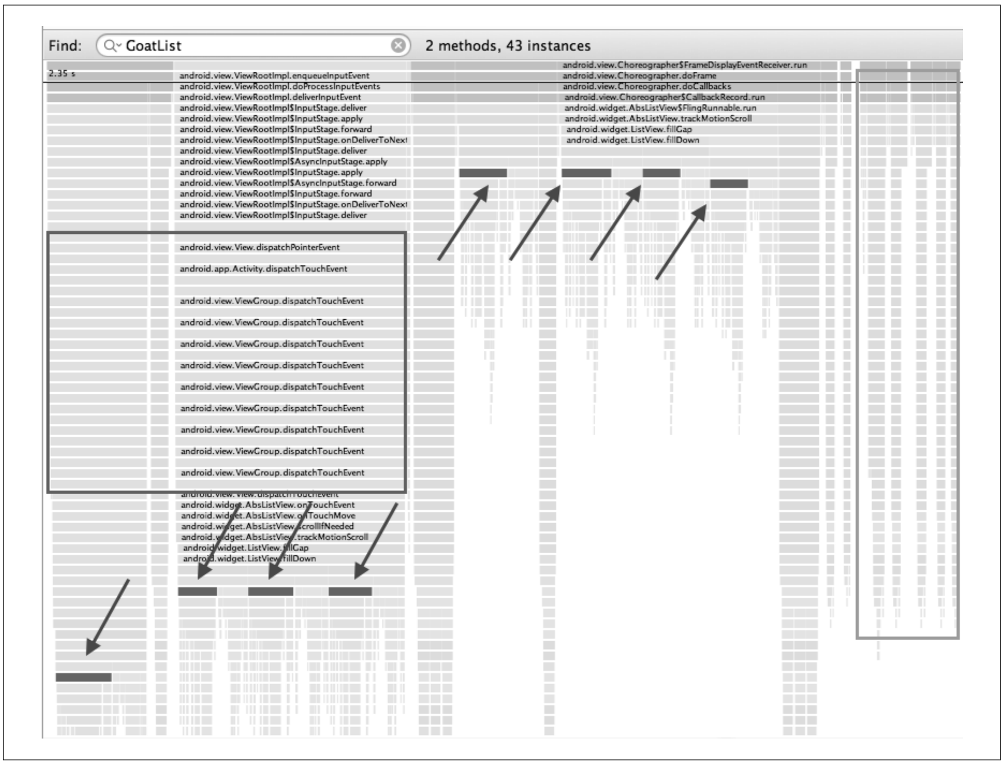


图 6-11：Android Studio 的 Traceview GoatList 过滤图

仔细看图 6-11，我们发现一件很有趣的事情，`GoatList` 在两个不同的上下文中被调用（根据纵轴的位置）。通过将视图从上端滑动到底部，在“慢 XML”视图中会生成这段轨迹。有四个 `GoatList` 视图在触摸事件（左边方框内）正表示“滑动”时被创建。后四个 `GoatList` 在我的手指离开（图像中心）之后快速滑动时被创建。当视图滑动到底部时，我们可以在右边方框中看见反弹动画的开始。

在第 4 章中，我们使用了 Hierarchy Viewer 阐释了平面视图层次的重要性。我们也可以在 Traceview 中作相似的分析。在图 6-12 的两张截图中，上面是“慢 XML”，下面是它的最优布局。这两张图的时间轴缩放是相同的，但是很明显，下面的视图（更优化的布局）填充视图更快（26 毫秒比 40 毫秒）。尽管两张截图类似，但在慢 XML 中，渲染每一项花费的时间和竖直的尖峰数量都要更多。对两种布局来说，顶部视图的渲染是很类似的，火焰图也体现出了相似的形状（方框①中）。这个视图创建时花费时间最长的是添加复选框（方框②），因为它有两种可能的状态，并且它在状态切换的时候还有一个动画。

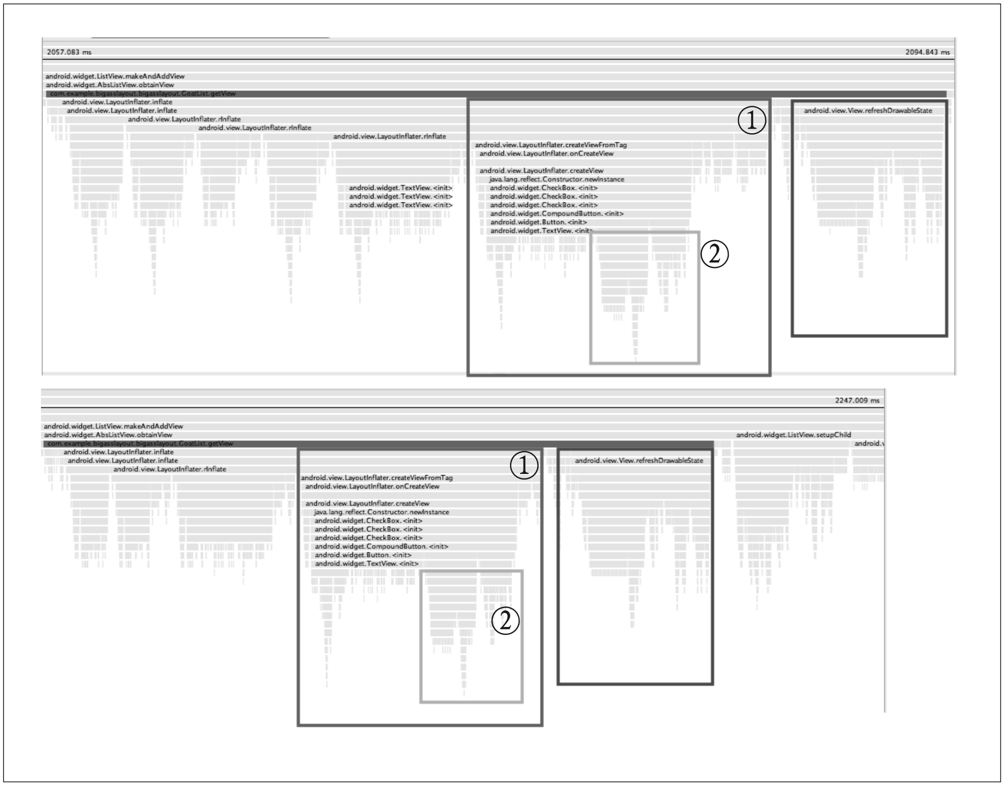


图 6-12：Android Studio 的 Traceview GoatList 比较图（顶部）；“慢 XML”视图（底部）：最大优化视图

在 `GoatList` 渲染完成后，由于需要检查复选框的状态，接下来还有一系列需要执行的指令（蓝色方框中）。对于那些“不是山羊”的行（未勾选状态），就不需要花费这 5~6 毫秒的时间。在这个 App 中，一共有 10 个勾选的行（山羊），3 个未勾选的行（不是山羊）。最初写这个 App 的时候，我把 13 个复选框全设置为选中，然后取消“不是山羊”的那三行。然而，我发现，Traceview 实际上将每个复选框都设置为选中状态（然后再以编程的方式取消选中），这给每个复选框都增加了“选中时间”，也就增加了 `GoatList` 填充布局的时间。根据 Traceview 的提示，我把它改为只检查那些“是山羊”的行（这样做也是必需的）。

现在，我们再来看看开启递归的斐波那契数列计算会发生什么，如图 6-13 所示。

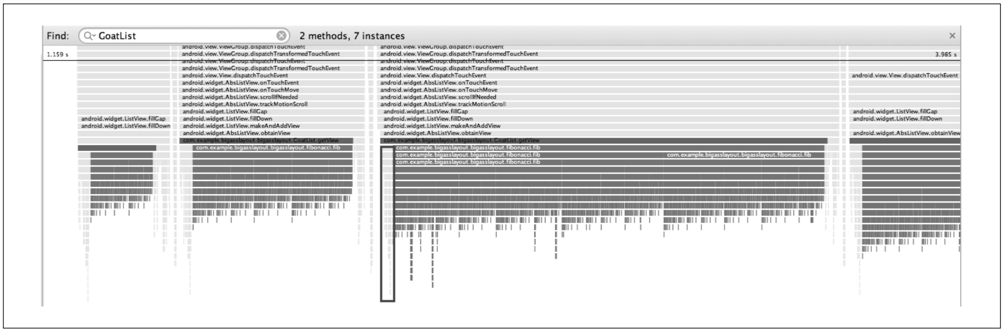


图 6-13：Android Studio 开启斐波那契延迟的 Traceview GoatList 图（另见彩插）

采集这段轨迹真是太艰难了，因为 Traceview 和递归斐波那契数列计算的开销导致我的设备出现了“无响应”错误。仔细看图 6-13，每一组火焰表示一行被绘制的过程（图中至少有 3 组，蓝色高亮的 `GoatList` 方法在每一组上方）。填充视图的过程在每一行都发生（蓝色的方框中显示的就是其中一次），但是斐波那契数列计算迫使 `GoatList` 方法挂起以执行这些计算（Traceview 用红色标记了这些导致 App 变慢的计算）。

当进行多线程测试的时候，我们可以轻松地比较特定时间内的 Traceview 图表来了解各个线程的状态。然而，Android Studio 提供的 Traceview 版本具有更好的兼容性，在查看特定时间内线程使用 CPU 的情况时有更好的表现。

6.5　其他优化工具

高通提供了一款免费 App——Trepn。通过这款软件，你可以查看内存、CPU、电量、网络和其他特性。在测试 App 时，它可以在屏幕的叠加层上显示这些信息。如果电话配备了高通处理器，还可以在测试时查看 GPU 使用情况。这些数据可以导出为 CSV 或数据库以供后续分析。然而，这些报告都是独立统计的，所以快速地比较 CSV 中的数据仍然是一件不容易的事情——最好将它们导入分析工具。

图 6-14 所示的方法可直观地查看 CPU 的使用情况。



图 6-14：Trepn 描绘的 CPU 概况图

6.6　小结

本章中讲解的工具都是免费的，并且为发现内存和 CPU 使用中的问题提供了大量信息。减少 CPU 使用，会让 `SurfaceFlinger` 和帧缓存确保每 16 毫秒生成一帧，从而提供流畅的体验，同时也帮助你节省内存和电量。

第 7 章

网络性能

智能手机最重大的改革之一就是将人类所有的知识存储到这个可以装进口袋的设备中。它可以帮助我们解答一些可能会被问到的重要问题。例如，“爸爸，长颈鹿的声音是什么样的呢？”我们也可以用它与世界各地的陌生人下棋或者玩其他类型的游戏。

随着网络吞吐量需求的增长，我们听说了各种言论，这些言论主要关于更快、更安全的网络是如何将信息存放得离我们更近的。在这里，我将打破幻想泡沫。尽管更新、更快的网络已经来临，但是距离 4G 网络的广泛使用还需要几十年的时间。在此期间，我们可以重点关注以下三点：第一，App 使用现有网络的方式；第二，网络的使用对于 App 运行的重要性；第三，网络对于设备电池的影响。正如我们在第 3 章中得出的结论：蜂窝无线网络、Wi-Fi、蓝牙这类方便信息交流的无线电也是导致电量耗损的主要因素。最大地优化设备中的网络性能可以使 App 更快地运行，同时减少消耗的电量。

在这一章中，我们将讨论以下三个问题：移动设备使用的不同无线信号的差异；用来分析 App 中网络使用情况的工具；一些可以取得巨大进步的简单的修改技巧。我们将研究如何在不同网络环境中测试 App（因为现在世界上大部分地区覆盖的还是 2G 和 3G 网络，必须确保 App 可以在这些环境中运行），最后看一下设备上的其他无线电：蓝牙、手表等外设间的通信和 GPS 定位扫描。首先，我们来看一下这些无线电是如何工作的，然后列出一些可以最大化其使用性能的方法。

7.1 Wi-Fi与蜂窝无线电

无线网络对比蜂窝无线电？与互联网的连接仅仅就是与互联网的连接吗？事实上，这两种

无线电的连接方式有很大的不同，根据 App 需要的数据量，你可能想建构两种不同的模型来进行内容下载：一种是蜂窝无线电，另一种就是 Wi-Fi。

当连接互联网的时候，有两个连接的属性会限制性能：带宽（管道的大小）和延迟（管道的长度或者管道的拥挤程度）。接下来看一下不同的无线连接是如何影响这两个属性的，以及在 Android 能耗统计文件（见 3.3.1 节）中的功耗值差不多的情况下，当蜂窝比 Wi-Fi 活跃时，蜂窝无线电是如何消耗更多电量的。

7.1.1 Wi-Fi

Wi-Fi 连接（在理想条件下）吞吐量比较大，是一种低延迟的连接，而且通常是不计费的（意味着使用 Wi-Fi 网络没有附加成本）。之所以加上“理想条件”，是因为你很少处在理想的无线网络环境中。因为 Wi-Fi 网络使用相同的频率，所以拥有多重 Wi-Fi 网络的区域将会交叠着有限的频率，导致所有网络共享带宽。

假设有一个没有带宽问题的 Wi-Fi 连接并且可以很好地连接到 Android 手机上。当 App 试图建立一个 Wi-Fi 网络连接的时候，会有一段非常短的延迟。当这种连接被建立的时候，无线电将处于高功率状态。一旦数据发生转换，无线将会停止使用。打开和关闭无线网络时会有一点延迟（测试结果为：开启的延迟时间为 80 毫秒，关闭时的延迟为 240 毫秒）。正如我们在 3.3.1 节中看到的那样，当 Nexus 6 连接上 Wi-Fi，且该手机处于待机模式时，消耗的电量为 3 毫安，而当其处于活动状态，并有数据传送时，所使用的电量为 240 毫安。考虑到 Android 设备电量的有限性，你就会明白为什么快速高效地下载是至关重要的。

Wi-Fi 具有高吞吐量、低延迟、没有数据收费的特点，这样 App 在使用 Wi-Fi 的时候就可以以一种更加“渴望数据”的方式运行。你可以使用更高质量的图片和视频，或许还可以与用户进行更多的互动体验。

7.1.2 蜂窝

目前，有多种不同的蜂窝技术在世界各地使用。用户连接的网络类型不同，其体验可能完全不一样。正如第 2 章讨论的那样，世界各地还有很多人仍然在使用 2G 和 3G 网络。见表 7-1。

表7-1：无线网络的演变

网络制式	代	最大下行速率（kbit/s）	延迟（ms）
GPRS	2G	237	300~1000
EDGE	2G	384	300~1000
UMTS	3G	2000	100~500
HSPA	3G	3600	100~500
HSPA+	3.5G	42 000	100~500
LTE	4G	100 000	<100

当 Android 设备连接到蜂窝数据网络的时候，为了维持两者的连接，所消耗的电量将随网络信号的强弱而变化。在 3.3.1 节中，你可能注意到我们可以使用两种模式来开启信号连接。当你处于强信号区域的时候，只需要使用低数值模式就可以进行连接，但是如果你处于蜂窝数据连接的低覆盖区，必须开启天线功率才能维持连接。当信号建立连接后，设备的电流从 4.5 毫安跳到了 125 毫安。虽然从原始数据上看，活跃的蜂窝无线信号（处于 125 毫安）比 Wi-Fi 信号（处于 240 毫安）耗电量低，但是鉴于蜂窝连接在网络中的实现方式，蜂窝连接的功耗通常会更高一些。为了最大限度地提高蜂窝网络的服务质量，所有运营商都采用了一种无线资源控制（Radio Resource Control，RRC）的状态机来控制数据连接的建立与中断。

状态机

状态机（有时是有限状态机）描述了拥有有限状态的事件的一种逻辑顺序。一种简单的状态机是具有开、关状态的灯具开关。而对于蜂窝网络而言，其网络连接的状态更多，并被用于优化其他因素，如网络、设备吞吐量、等待时间和电源功耗等。

7.1.3 RRC状态机

当手机启动数据连接后，在创建 TCP 连接之前，会有几种初始无线信号被发送到信号塔。这些信号会使无线连接的创建时间增加 500~1000 毫秒。延迟是移动连接的关键方面之一，而状态机试图抵消这种延迟。

每一种移动网络都有一个 RRC 状态机，用于确保在最后一个数据包发出之后无线电信号仍然开启，以补偿建立连接时产生的延迟并平衡电量消耗。每个载波可以指定不同的状态以及设备在该状态停留的时间（正因如此，每种网络都稍有不同）。世界各地的每种网络具有不同的特定变量，因此掌握精确的时间并不是很重要，但是掌握好 RRC 状态机的基本原理对理解蜂窝连接的操作方式大有裨益。知道了这一点，你就能够优化网络连接，使之与 RRC 状态机协同工作，从而帮助 App 更快地运行，并减少电量的使用。

状态机注意事项

讨论（或列出）所有移动运营商使用的计时器（哪怕是同一个运营商，计时器也会因地区而异，并且会随着时间而改变）远超出了本书的范畴。作为开发者，优化 App 同每一个运营商的 RRC 状态机的连接是不切实际的。重要的是要理解状态机是什么，以及状态机的存在对移动 App 有什么危害（或帮助）。

此外，3G（GSM、CDMA）网络和 LTE 网络所使用的状态机是不同的。简单起见，这里

介绍一下 LTE RRC 状态机。

1. 4G（LTE）状态机

当数据将要发送时，Android 手机将从空闲状态（低功率消耗）转变到连接状态（高功率消耗）。当停止发送数据包后，无线连接并没有立即关闭，而是有一个 10~15 秒的高功率延续时间段。一旦无线连接立即关闭，如果接下来仍有后续数据快速地发送过来，这些数据将不得不再次跨越高延迟的连接时间段，这种发送、接收数据的方式会使用户体验变得缓慢。正因为无线连接仍然处于连接状态，后续发送的数据包延迟时间将大大减少，数据包将会被快速地运输。当最后的高功率延续时间段没有数据包时，无线连接将会关闭以节省电量。

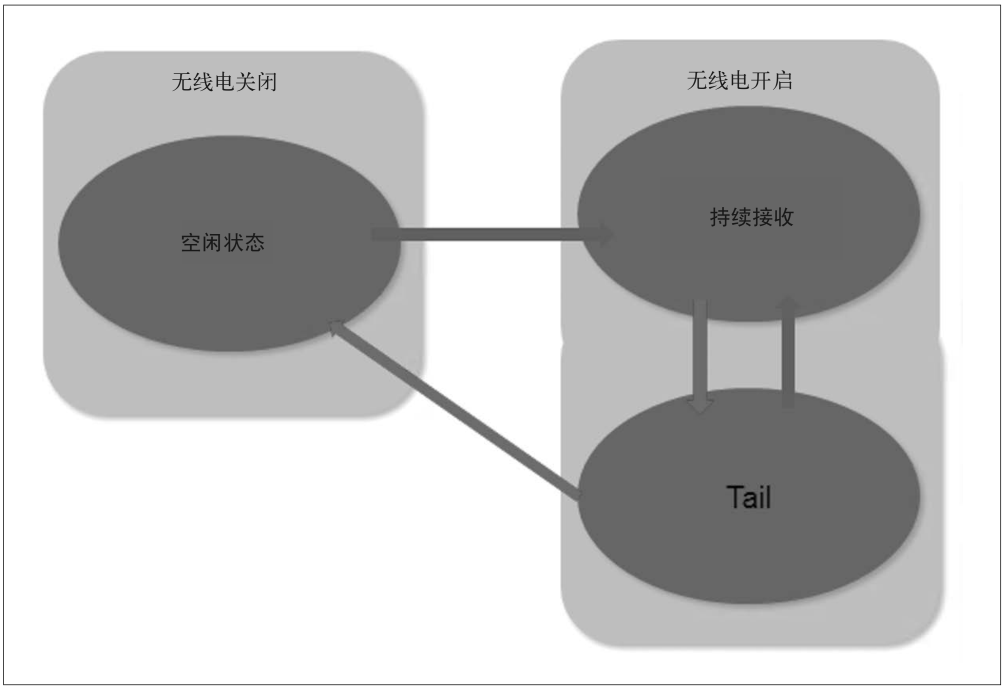


图 7-1：LTE RRC 状态机

如果再看一下表 7-1，你会发现，随着网络的不断改善，延迟在逐渐降低，同时网络的吞吐量也在逐渐增加。4G 网络规范已经大大改善了建立数据连接（RRC 从空闲状态向连接状态的过度）所需要的信号。与 3G 网络相比，4G 网络连接建立时的延迟时间降低至原来的 10%~20%（如表 7-1 所示）。曾经在 3G 网络中是 300~1000 毫秒，现在在 LTE 中是 50~100 毫秒。尽管带宽中 4G 网络的改善令人印象深刻，但真正使 LTE 像它展示的那样快的原因是延迟的改善。Ilya Grigorik 在他的作品《高性能网络浏览器》中使用大量篇幅描写了延迟的影响力。另外他还更加细致地介绍了所有的网络性能。

一般情况下，LTE 无线连接比 3G 无线连接耗费更多的电量。但如果正在传输的文件很大，LTE 的下载速度可能要更快，结束无线连接的时间也会相应缩短，因此使用的电量更少。但是，绝大多数的移动数据传输并不是大型文件，而是由数百个小型文件构成，因此使用的数据块也相对较小。这些小小的文件无法充分利用 LTE 的带宽能力（因为它们太小了），因此通过 LTE 下载内容所耗费的电量比使用 3G 网络要稍多一些。

无线电连接和数据连接

无线电连接和数据连接之间存在细微的差别，我们在此讨论一下。信号塔和手机之间的物理无线电连接不同于手机和服务器之间的数据连接。数据连接基于无线电连接之上，也就是说，传输数据前必须先建立无线电连接。打个比方，无线电连接就像一座升降桥，而数据连接就是桥上的马路。

如果数据连接处于连接状态，但是目前并没有在传输数据，只是为将来的需要做准备，那么手机和信号塔之间的无线电连接可以暂时挂起（省电）。回到我们刚才的升降桥比喻，也就是说，马路还在，但是升降桥暂时处于吊起状态，让出通道以便桥底的船通过。如果服务器向设备发送数据，信号塔就会向设备发送无线电信号以重新建立无线电连接，并允许数据连接的完成（放下吊桥，允许汽车在马路上穿行）。

这听起来很棒，我们会想，为什么不干脆让所有的连接都保持开启状态以备将来使用？由于蜂窝网络的连接数量有限，一段时间后（通常是 5~30 分钟），网络会自动清理孤立的连接（估计网络开发人员是想拆掉闲置的升降桥，改建河畔公寓）。

2. 你的App在使用RRC状态机吗

如果你在使用 RRC 状态机，那么网络连接所消耗的电量绝对超出了你的想象。分组连接并确保无线电使用时间最小化，可以极大地提高移动 App 的性能。所有的数据连接在传输数据时至少需要 10 秒（相当于 5 分钟的待机时间）。一直以来，很多人都认为数据下载速度是移动 App 的性能关键。但是很显然，移动 App 除了下载速度要快以外，尽可能少地开、关无线电通信也是节省资源的关键。

手机通话时使用数据

我们生活在一个多任务化的世界。使用 App 的过程中，打电话是很常见的现象。如果 App 是通过 LTE 连接的，手机来电时会降到 3G 数据连接以完成剩余的数据会话。

这是由电路交换回落导致的。在 Voice over LTE（VoLTE）成为标准规范前，所有的语音通话传输都是通过 3G 电路交换网络。这就意味着，正在活动的任何数据会话都会降到 3G 网络。在 3.5.3 节中，我们讨论过一个研究——我在通话的同时进行了电话会议。仔细观察通话时间（见图 7-2），我们发现无线电连接一开始是浅色（LTE），但是在通话过程中变成了深色（HSPA）。通话结束时，无线电又转回了 LTE 数据网络（浅色）。

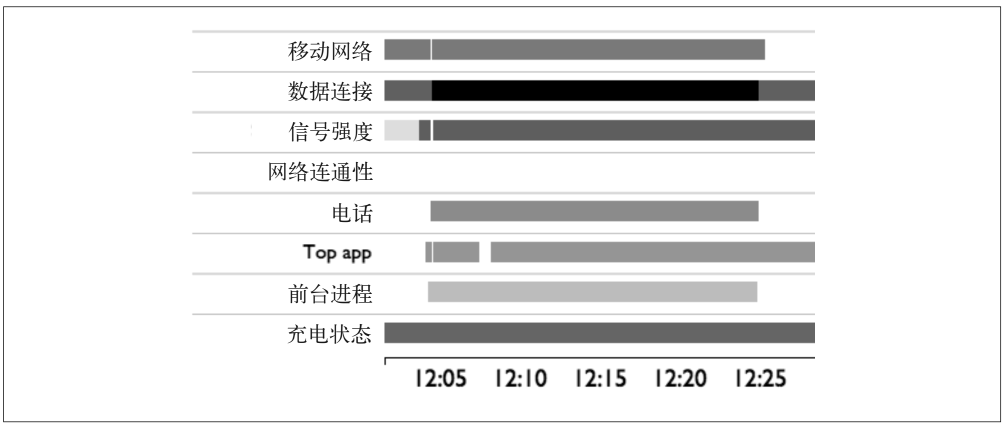


图 7-2：Battery Historian 显示电路交换回落

7.2 测试工具

到目前为止，我们已经讨论了 Android 无线电的耗电问题，以及移动数据网络的运作方式。那么我们该如何使用这些知识来优化 Android App 的流量呢？再者，即使优化了流量，又如何进行测试和确认呢？我们有大量的工具可以捕获和分析移动流量数据。多年来，世界各地的网络运营人员都在使用 Wireshark 和 Fiddler 之类的工具收集封包数据，并分析其中的潜在问题和优化方案。中间人（MITM）工具帮助用户解密 HTTPS 流量，以便用户了解哪些数据可以在网络上安全发送。这些毫无疑问都是测试移动 App 性能的一线工具。AT&T 公司开发了一款类似的工具——应用程序资源优化工具（Application Resource Optimizer，ARO），它可以捕获数据包，还可以为 App 开发人员提供移动通信流量的优化建议。

7.2.1 Wireshark

Wireshark（https://www.wireshark.org/）可能是世界上应用最普遍的网络分析工具。它是一款免费的桌面运行工具，用于收集在数据连接上传送的网络封包，可以实时监测数据，并将数据收集起来存储于文件夹中用于进一步分析。

要使用 Wireshark 测试 Android 手机，必须将手机连接到已安装 Wireshark 的 PC 上。如果是 Windows 机器，我使用 Connectify（http://www.connectify.me/）将笔记本电脑转换成 Wi-Fi 热点，然后把 Android 设备连接到热点，所有的数据流量开始从手机传输到电脑（可以被 Wireshark 监测到）。到这一步，你可以开始在 Wireshark App 中通过无线接口收集数据了（如果你不确定哪个是 Wi-Fi 接口，启动手机的网络流量就会看到有一个网络接口开始发送和接收数据包流量）。

在图 7-3 中我们可以看到，Wireshark 显示了手机（192.168.223.104）和电脑（192.168.223.1）之间来回发送的每一个数据包。一开始要弄明白其中的事件有点困难，所以我添加了一个 HTTP 过滤器，这样就能把数据包限制在 HTTP 数据包的范围内，然后我再观察测试过程中产生的请求和响应。现在你可以看到我在浏览器里打开 cnn.com 发起请求。数据包 1481 显示我的 cnn.com 请求给 www.cnn.com 引起了一次 301 重定向（方框中），该页面向 CDN 催生了大量文件请求。如果我想知道这次抓包中产生了多少个 301 重定向，可以使用“http.response.code == 301”过滤，并查出三个这样的重定向。

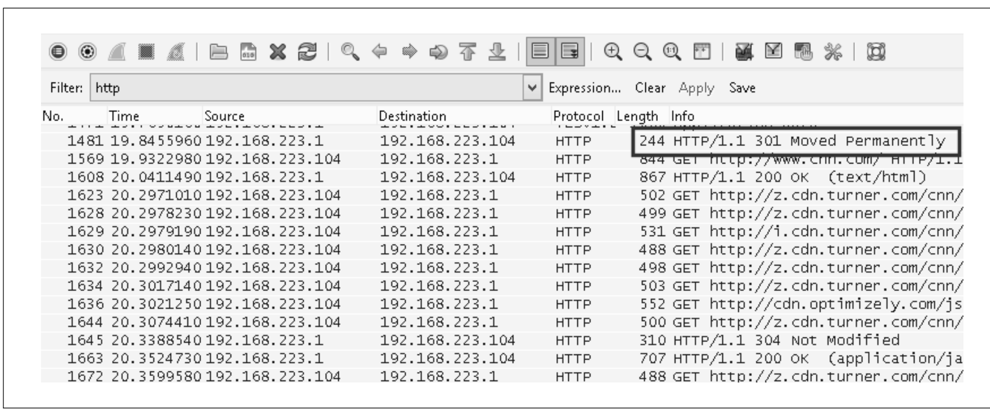


图 7-3：使用 Wireshark 抓包

Wireshark 里的过滤工具非常强大，该工具可以搜索和过滤特定文件、特定类型的文件、带有缓存头的文件等（可能性数不胜数）。在实践中，这些搜索都很强大，但是你必须要有针对性的问题，否则这种方法用起来就像大海捞针。相反，如果你对问题有了大概的了解，Wireshark 可以很好地帮助你深入挖掘并准确定位问题。

7.2.2 Fiddler

Fiddler 是另一款分析网络流量的免费工具，为设备的所有传输数据充当代理。作为代理，Fiddler 还可以充当 MITM 工具，允许用户解密 HTTPS 流量。在 Wireshark 中，你可以看到正在传输的 HTTPS 流量，但是你无法通过解密来获取文件类型或文件内容。

Fiddler 的使用方式和 Wireshark 类似：把 Android 设备连接到 Connectify Wi-Fi 热点，然后通过更改设备上的 Wi-Fi 设置添加 Fiddler 代理，在设备和 PC 上安装 Fiddler 证书，最后，Fiddler 就能够读取所有的传输数据了（Fiddler 的网站上有完整的安装说明）。

完成所有连接后，你可以在 Fiddler 应用窗口里看到流量传输信息。如图 7-4 的屏幕截图所示，我正在使用 YellowPages App 查找周边的杂货店。

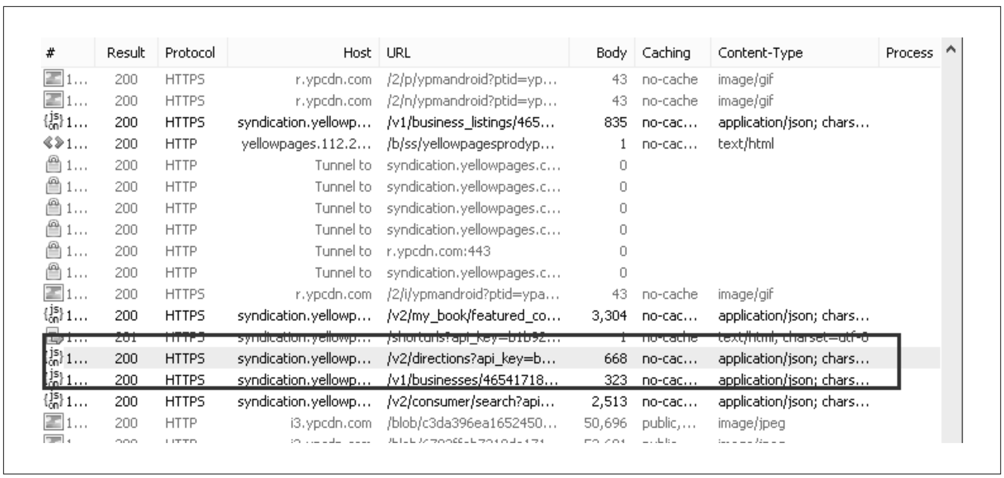


图 7-4：使用 Fiddler 代理抓包

我们看图 7-4 中 Fiddler 抓包（Fiddler packet capture）的左侧窗口，方框区域的字段是一条来自 YP App 的响应，文件是 668 个字节（连接中），缓存设置为 no-cache，并且是一个 JSON 文件。它包含了我家到杂货店的所有路线，而且使用了 HTTPS 加密（非常好，因为响应文件里有我的地址和位置信息）。Fiddler 窗口的右侧（见图 7-5）有很多选项窗口。顶部窗口显示发送到 syndication.yellowpages.com 的请求标题，底部窗口显示被解密的响应。在 JSON 文件里，你会发现到杂货店的总距离是 4.787665 英里（真是相当精确）。App 还进一步预测了开车到杂货店的时间是 661 秒，或者说是 11 分钟多一点。

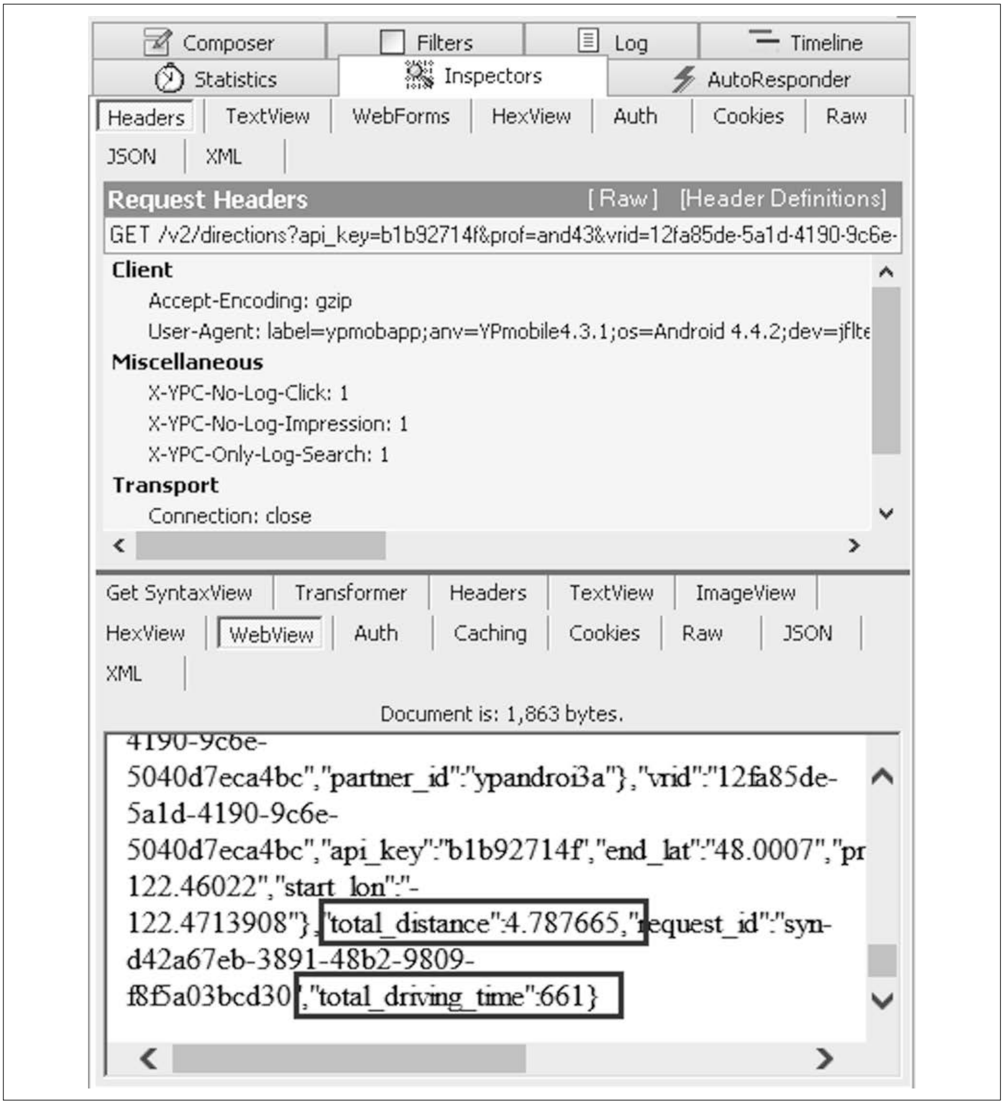


图 7-5：Fiddler 代理抓包的窗口详情

7.2.3 MITMProxy

MITMproxy 是一款类似于 Fiddler 的代理工具，通过创建一个 MITM，解密当前网络上传输的 HTTPS 数据。

解密 HTTPS 流量非常有用，因为很多数据流量都是使用 HTTPS 来保护用户的数据。通过解密数据，Fiddler 和 MITMproxy 也可以确保你向任何已添加到 App 的第三方 SDK 发送正确的文件和信息。

7.2.4 AT&T ARO

ARO 是专为 Android 和 iOS App 设计的一款网络性能监测工具，由 AT&T 开发，免费开源。它包含很多与 Wireshark 以及 Fiddler 相同的数据包捕获功能。但是又与 Wireshark 和 Fiddler 不同（二者需要通过 Wi-Fi 连接到计算机），ARO 能够收集蜂窝网络的数据包跟踪信息。一旦 ARO 从测试中收集到数据，它会对数据进行处理，并为开发者提供友好的图表信息以帮助开发者更好地分析数据。而且 ARO 会按照移动网络的 25 个最佳实例测试流量，并立即向测试者反馈哪些领域的性能可以改进。图 7-6 是测试总结（叉号表示失败，勾号表示跟踪通过测试标准）。在本章中我们将讨论这些最佳实例。

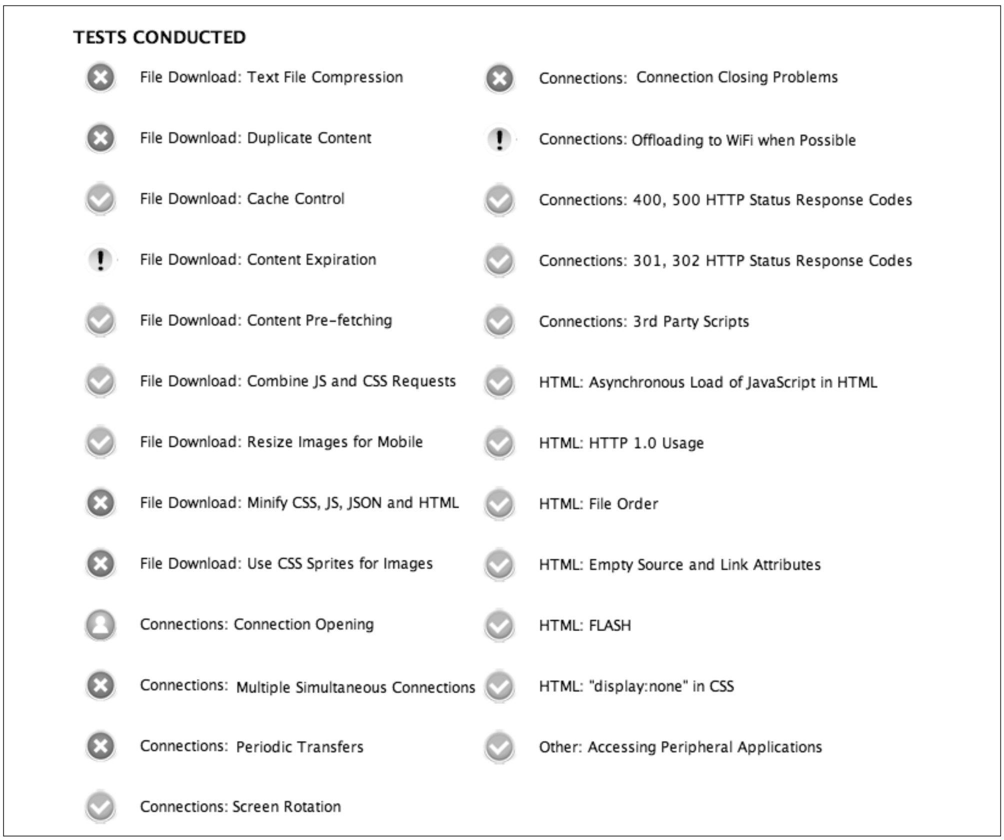


图 7-6：ARO 最佳实例：通过 / 失败

适用于 Android 设备的 ARO 有两个版本。ARO 数据收集器 APK 直接在设备上运行 TCPdump 以收集所有的数据包（并为每个连接分配进程）。这有赖于 Android 设备具备 root 权限，为了简化测试，我们也可以使用不需要 root 权限的版本。在没有取得 root 权限的情况下，我们无法将连接分配至特定进程，以至于将流量锁定至特定 App 会更加困难（如果设备上运行很多 App）。

一旦在 ARO 中跟踪采集到一个数据，就可以在 ARO 分析工具里分析。App 中执行的测试会根据图 7-6 所示的 25 个最佳实践被评测。每个最佳实践后面列出了更多的详细信息，如果没有通过任意一项最佳实践，你可以得知失败的缘由（见图 7-7）。

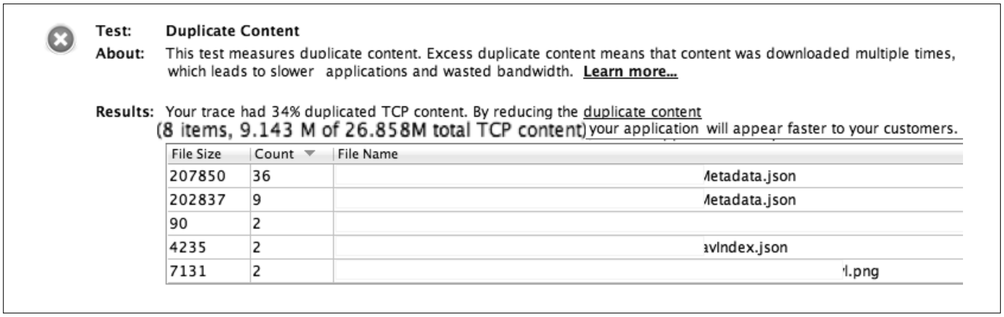


图 7-7：ARO 关于重复内容的最佳实践；34%（9MB）的数据被多次发送，太多了（为方便阅读放大了一些文字）

每项数据跟踪都提供了 5 个选项卡，但是只有在诊断（Diagnostics）选项卡下你能真正看到数据是怎样流入和流出 App 的（见图 7-8）。

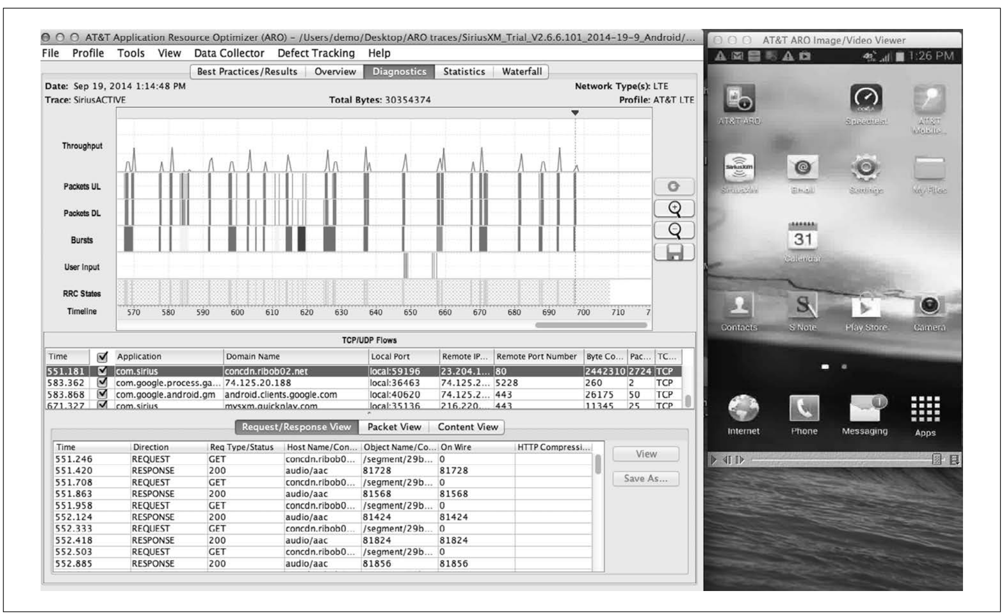


图 7-8：ARO 诊断选项卡

诊断选项卡包含很多信息，让我们看看这里给出的数据。显示的有两个窗口：左边的是数据分析，右边的是跟踪过程中的屏幕录像。录像与跟踪同步，所以当你选择一个数据包或连接时，可以看到那个时刻屏幕上显示的内容。

仔细看图 7-9 中的诊断选项卡，图中记录了随时间变化的数据包流量。最上行显示的是随时间变化的正常吞吐量。通过对比可以看出哪些连接的流量较大，哪些连接的流量较小。下面两行显示的是连接过程中上传和下载的数据包。标有 Bursts 的那一行用不同颜色表明了突发流量类型。红色的是由 App 发起的，黄色的是由服务器发起的，绿色的发生在用户输入事件后（在下一行记录），蓝色突发流量的发生通常是由于连接关闭后，产生了空数据包。最下面一行表示 LTE RRC 状态机，如图 7-1 所示。纯色表示连续接收，网纹区域代表 Tail。空白区域代表无线网络关闭（Idle）的时间。请注意蓝色箭头和蓝色虚线。这代表的是录像查看器上显示的跟踪时刻。如果你点一下录像的播放按钮，会看到这条线向右移动，然后你可以看到正在传输的数据包，同时还可以看到设备屏幕上显示的内容。

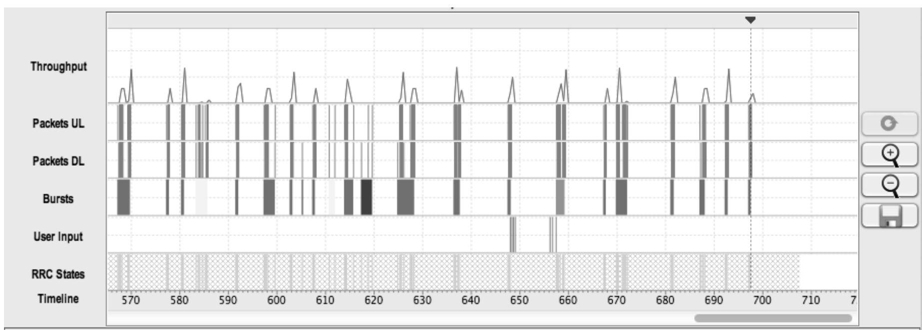


图 7-9：ARO 诊断选项卡数据图（另见彩插）

图下方的两个表为网络跟踪过程中发起的每项数据连接提供了更多信息（见图 7-10）。上面的图显示了跟踪过程中启动的每一项 TCP/UDP 连接。当前选中的是 SiriusXM Radio App 在 551 秒时启动的连接。你可以看到域名和 IP 信息，以及字节计数和连接产生的数据包数。

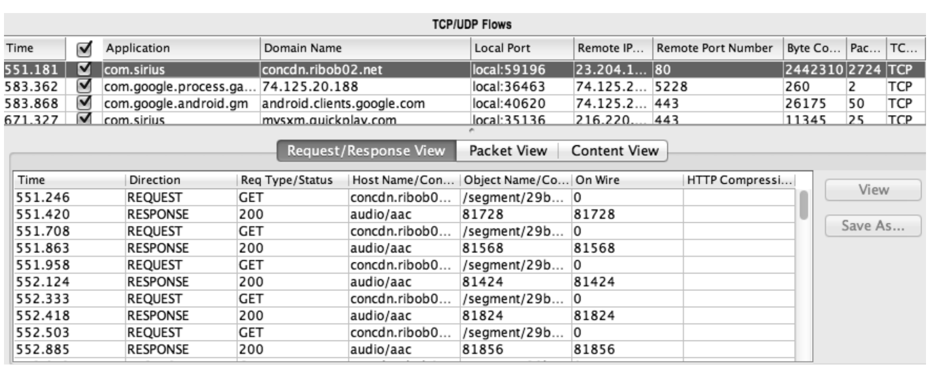


图 7-10：ARO 诊断卡数据表

下表显示了上表选中的 TCP 连接发起的请求和响应，该表显示，此连接传输了很多 81KB

的音乐文件。

ARO 中还有一些视图也提供了可观的信息量，在本章讨论潜在优化方案时，我们还可以看到更多的屏幕截屏。

ARO 的缺陷就是它无法解析通过 HTTPS 发送的任何文件的详情。如果 App 使用的是 HTTPS，你必须在 Fiddler 跟踪中手动查看这些文件。

7.2.5　混合型App和WebPageTest.org

WebPageTest.org（http://www.webpagetest.org/）是一款强大的网站测试工具。你可能在想："这本书是关于 Android App 开发的，为什么你要讲网站测试的内容呢？"成千上万的 Android App 都是使用 PhoneGap 之类的工具开发出来的，这些工具只是简单地将来自网络的组件用原生代码包装起来，提供一种更加原生的应用体验。这种原生 App 通常只在一个嵌入式的 Web 视图中显示内容，使它表面上看起来原生化。因为不能优化 wrapper，所以作为一个混合型 App 开发人员，你唯一能做的就是确保 Web 组件可以最快地运行。

通过 WebPageTest，你可以在全球几个不同的地方测试网站，但是在弗吉尼亚州的杜勒斯只有几台 Motorola 和 Nexus 设备可以用于测试（带有 Chrome 和 Chrome Beta）。WebPageTest 测试可以显示网页在手机上的加载速度并指出需要改进的方面。

7.3　Android网络优化

Web 性能社区有许多关于网站的最佳实践，这些最佳实践同样适用于移动 App。我们将讨论几个 Android App 的最佳实践（它们也适用于你可能会做的所有 iOS 开发）。这里列举的最佳优化实践的重要性不分先后，因为每一项都会对 App 性能产生不同的影响（有些可能根本不适用于你的 App）。网络性能优化的总体趋势（无论是台式机还是移动终端）是以最快的速度下载任何东西。关闭无线电通信可以节省电量。尽可能以最快的速度将内容发送给用户，从而提升用户体验。

移动 App 性能优化的基本原则基本上源自于 Steve Sounders 在《高性能网站建设指南》中所列的 14 条标志性性能优化原则：

- 减少 HTTPS 请求
- 使用内容发布网络
- 添加 `Expires` 头
- 压缩组件
- 将样式表放在顶部
- 将脚本放在底部
- 避免 CSS 表达式

- 使用外部 JavaScript 和 CSS
- 减少 DNS 查找
- 精简 JavaScript
- 避免重定向
- 移除重复脚本
- 配置 `Etags`
- 使 AJAX 可缓存

这里面有些原则是针对网站的，但是大部分还是适用于 Android 本机优化，我们会在以下几节内容中具体谈到这些原则。

7.3.1 文件优化

有两种基本方法可以更快地下载数据：减少请求次数（Sounders 的第一条原则），或者是减小这些请求的大小（Sounders 的 14 条原则中若干条建议与此类似）。因为 App 变得日益复杂，所以这并不是一件容易的事情。但庆幸的是，本章中的建议可以帮助你找到一些方案来减少 App 内容的数量并缩减其大小。

压缩文本文件（压缩组件）

这是最简单的解决方式之一。当向 App 发送文本文件（HTML、CSS、JavaScript、JSON，等等）的时候，在服务器上压缩文件可以将文件减小到原来 12.5%~20%。大幅度压缩服务器向 App 发送的文件，意味着文件的传输次数更少而且发送速度更快。例如，我们在 App 中下载了一个没有进行过 Gzip 压缩的、大小为 200KB 的的文本文件。若将它放到已启用压缩的服务器上，可以将其在线大小减少到 51KB。这样不仅可以把文件内容更快地发送给用户，还可以节约服务器带宽并减少损耗！

如今可用的 Gzip 算法有很多。一般情况下，大多数 App 采取标准的 Gzip 压缩就足够了。如果你真的想最大限度地压缩，而且文件又不需要经常更改，你可以试试 Zopfli 压缩算法。与默认的 Gzip 算法相比，它的压缩率要高出 5%。但是它有个缺点，它的文件压缩执行时间要多出 100 倍（因此警示大家“只用它处理预压缩过的文件”）。

启用 Gzip 压缩只需要对服务器做一个简单的更改（不需要更改 App 代码），你只需要给 .htaccess 程序代码文件添加文件扩展名 /MIME 类型：

```
<ifModule mod_gzip.c>
mod_gzip_on Yes
mod_gzip_dechunk Yes
mod_gzip_item_include file .(html?|txt|css|js|php|pl)$
mod_gzip_item_include handler ^cgi-script$
mod_gzip_item_include mime ^text/.*
mod_gzip_item_include mime ^application/x-javascript.*
mod_gzip_item_exclude mime ^image/.*
```

```
mod_gzip_item_exclude rspheader ^Content-Encoding:.*gzip.*
</ifModule>
```

你立即就能看到文本文件的下载速度变快了。另外还可以给 Gzip 增设排除小文件的功能，把不足 850 字节的文件放到一个可以不用压缩就能发送的数据包里。当然，在 Gzip 压缩 / 解压负荷很小的情况下，你也可以省略这些过滤小文件的步骤。

ARO 对跟踪过程中捕获的所有文本文件进行 Gzip 压缩测试。你可以在 ARO 中的两处地方发现文件是否被压缩了。

- 最佳实践：文本文件压缩

  所有未经压缩便被发送的文本文件都会列在文本文件压缩（Text File Compression）中（见图 7-11）。目前，还没有办法直接获取存储文件以添加压缩，但是该表如实报告了未压缩文件的大小。注意，不足 850 字节的文件都没有标记出来（因为它们在不压缩的情况下就能装入一个数据包，从而一次传完）。

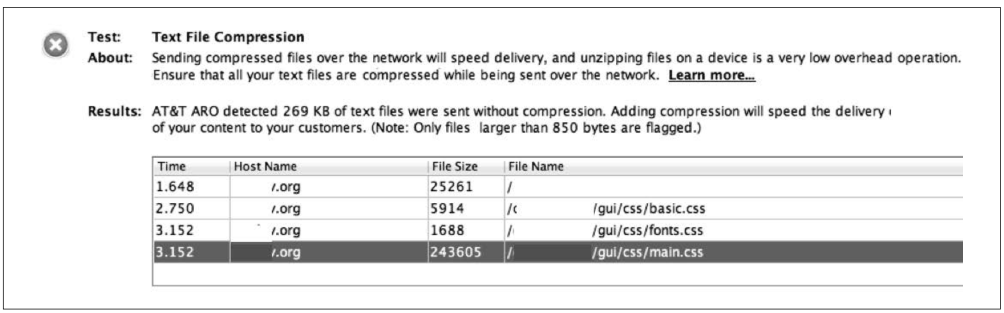


图 7-11：ARO 文本压缩最佳实践

- 诊断选项卡（请求响应表）

  图 7-8 中，最下面的表格表示请求和响应。最右边的一列可以识别压缩文本文件，如果没有压缩，显示的就是“none”。

7.3.2　精简文本文件（Souders：精简JavaScript）

精简是另外一个缩小文本文件的方式。精简过程就是去掉文本文件中所有只具有方便阅读作用的格式（比如空格、制表符和注释），进而使文件变小。 例如：

```
<html>
 <title> A Sample Page</title>

<body>
with some sample text
<--do more here-->
</body>
</html>
```

修改为：

```
<html><title> A Sample Page</title><body>with some sample text</body></html>
```

根据页面大小和复杂度，精简可以将文件缩小 20%~50%。很多构建工具（如 grunt）自带精简库，在你做改动的时候就可以自动精简文件（帮你节省了很多工作）。

有人可能会说，使用 Gzip 就足以降低文件文本的传输成本。例如，精简虽然可以把文件缩小 10%~15%，但是精简过和未精简过的文件进行 Gzip 压缩后，所节省的文件大小差异只有 1%~2%（因为空格的压缩效果好）。

可是，即便只能节省 1%~2% 的网络传输量，你也应该养成精简的习惯，因为这样能为用户节约设备存储空间。另外，文件越小，读入内存的速度就越快（而且导致内存有限的设备崩溃的可能性越小）。

虽然 Sounders 的原则只考虑到精简 JavaScript，但实际上这个优化策略可以用于缩小任意文本文件。ARO 工具把所有的 CSS、JavaScript、JSON 和 HTML 文件都考虑在精简范围内了（见图 7-12）。它不仅把每个文件的潜在节省量计算了出来，而且还统计出了跟踪中捕获的所有文件所能达到的总节省量。

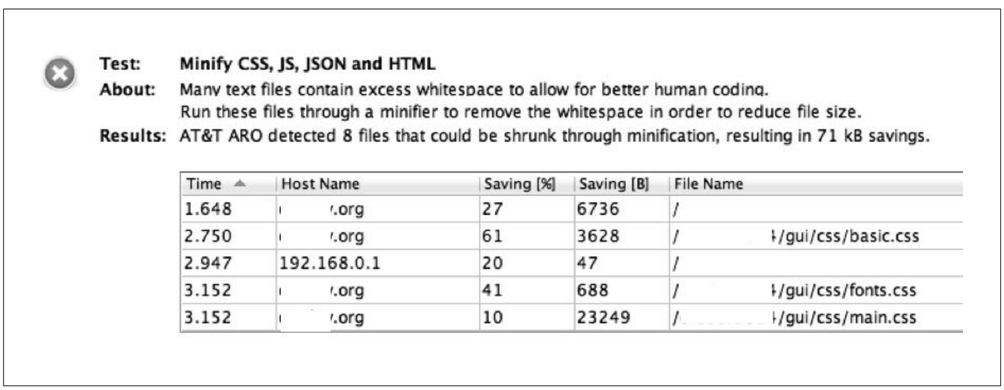


图 7-12：ARO 精简最佳实践

7.3.3 图片

图片是 App 中最常见的文件下载类型，同时也是最大的一种文件形式。它无处不在，易从网页或者其他数字服务中获取。控制图片大小可以有效减少 App 的数据流量。App 必须在图片质量和图片尺寸之间找到一个平衡点（可以咨询用户体验或编辑团队）。一旦找到了正确的平衡点，你就拥有了一个外形美观的 App，而且优化后的图片可以快速下载和呈现。

1. 尺寸超大化

如果 App 里的每张图片只有一个版本，要将它用于所有移动设备（包括使用其他移动操作系统的视网膜显示屏平板电脑），那么你可能要使用一张在所有设备上看起来都很清晰的图片（这意味着下载的图片非常大）。想象一下将适用于视网膜显示屏平板电脑上的一张图片通过 2G 网络发送到小小的 Android 设备上需要多长时间，你就会明白这不是你想要的用户交互。

为了适用于各种不同尺寸的屏幕，你可能想要为图片创建图片 bucket。这样在需要一张图片时，App 就可以提供屏幕尺寸以确保传递尺寸正确的图片。Android 为 App 开发提供了屏幕分辨率 bucket，这些值对网络图片而言也是一个良好的开端。

如果缩略图和原图在 App 上的展示效果一致，就可以考虑下载缩略图而不是原图。

> **缩略图**
>
> 我以前用过一款流行 App，它的每篇文章旁边都有一个 250KB 的缩略图。一个页面上有 6~8 个文章标题，每次启动 App 的时候，这些小图片就要增加 1.5~2MB 的数据。由于尺寸原因，开发者把它们戏称为缩略图（dumbnails），在后来发布的版本中采用了 5~10KB 的小图片代替它们。

你可以选择的图片尺寸很大程度上并不是由 App 决定的，而是由布局得知屏幕上必须显示多大的图片决定的。因此，只有先根据中小型平板设备和手机屏幕的大小计算出最合适的像素尺寸，才可以为 App 创建正确的图片 bucket。

2. 元数据

当用数码相机拍照片时，文件里很可能有与照片相关的元数据（包括设备名称、设备设置以及拍摄地点等信息）。照片编辑软件也会给图片增加一些元数据。除非 App 是讨论照片拍摄方法和编辑技巧的摄影 App，否则你可以去掉照片上所有相关的元数据，将这些几字节到几十千字节不等的元数据保存在其他地方，这样不会损坏发送给用户的图片的质量。

3. 压缩

跟文本文件一样，你也可以压缩图片使其变小，从而减少占用的设备存储空间，同时缩短下载时间。图片压缩这个话题太大，在这里不便详述，但是必须指出，在压缩图片的同时，图片质量也大幅下降了（有损压缩）。

图片适用的有损压缩率取决于图片的用途。对缩略图而言，压缩率可以更高，因为它们本身就很小，很难发现有粒子或像素变化。对于文章中的图片，以 70% 的压缩率保存 JPEG 就足够了。专注于摄影与图形的 App，可以选择不做任何有损压缩。压缩率是优化图形质量与为速度压缩尺寸之间的一种微妙平衡。 Google 的 PageSpeed 服务器默认的图片压缩率是 85%，这可能是图片对比的好起点。

WebP：能否成为 JPEG 的接替者

WebP 是 Google 正在开发的一种图片格式。WebP 格式图片通常比类似的 JPEG 格式图片小 20%。支持 WebP 的浏览器和设备日益增多（Android 4.0 及更新版本支持）。在降低图片文件大小方面，WebP 值得考虑。

7.3.4 文件缓存

如果 App 中存在经常使用的文件，那么你应该一*次性*下载好这些文件并保存在本地以供多次使用。就性能而言，本地读取文件往往比建立连接并下载文件更快。单凭这一点，缓存就可以加速 App 的呈现。减少网络连接不仅能节约服务器容量，还能降低用户的电量消耗。

当然，调用缓存的主要原因是移动数据的使用受流量套餐限制，下载过多内容可能会使用户超出每月流量限制，增加花费。缓存有两个要点：首先，必须在设备上的 App 内打开缓存功能，其次还必须在服务器端设置正确的缓存时间。

1. 应用内缓存

有趣的是，Android 缓存功能默认是关闭的，所以你需要将它打开。对于 Android 4.0 及更新版本，在 `onCreate` 中调用以下代码就可以启用 HTTP 响应缓存：

```
private void enableHttpResponseCache() {
  try {
    long httpCacheSize = 10 * 1024 * 1024; // 10 MiB
    File httpCacheDir = new File(getCacheDir(), "http");
    Class.forName("android.net.http.HttpResponseCache")
         .getMethod("install", File.class, long.class)
         .invoke(null, httpCacheDir, httpCacheSize);
  } catch (Exception httpResponseCacheNotAvailable) {
    Log.d(TAG, "HTTP response cache is unavailable.");
  }
}
```

现在 App 可以缓存了。

2. 服务器端缓存

服务器向设备传输的报头设定了设备里存储的每个文件的缓存时间。在服务器上设置缓存参数时，必须考虑几个重要因素。通常，文件缓存有一个有效期，App 在有效期内请求文件，由设备缓存提供该文件。如果缓存过期，则连接服务器，检查文件是否被修改。如果文件相同，则向设备发送一个 HTTP 304“not modified”响应，并重置缓存定时器；如果文件不同，则下载新文件。

缓存定时器的长度实际取决于缓存内容及其改动的频率（例如，运动队标志这些很少改动的可以缓存一年，天气情况缓存 5 分钟，头条新闻订阅源可能不缓存）。更改内容的缓存

时间不仅可以确保用户看到的数据始终是最新的，还能限制重复下载的文件的数量，从而节省电量和流量。一般来说，有三种报头可以用来设置内容的过期日期。

3. Cache Control（添加Expires报头）

缓存最常用的报头是 `Cache-Control` 报头，它有几个可以指定的常用值。

- Private/Public

  网络中 CDN 缓存使用的代表性指令。它可以向 CDN 表明文件是公共的（任何人都可以使用）还是用户私有的。

- no-store

  如果文件使用该指令，那么该文件无法缓存，必须在每次使用时下载。

- no-cache

  no-cache 报头的名称可能令人误解。带有 no-cache 报头的文件实际上是可以缓存的，但是再次使用前必须重新验证。

- max age=X

  max-age 表示文件可以缓存的最大时间（单位：秒）。常用值为 0（与 `no-cache` 相同）、60、300、600、3600（1 小时）、86 400（1 天）、3 153 600（1 年）。

4. ETag

`ETag` 是一种响应报头，包含由随机字符组成的唯一字符串。每次从缓存中使用文件时，`ETag` 必须先在服务器端验证。如果本地字符串与服务器端一致，服务器发回“304 not modified”，那么使用本地文件。如果 `ETag` 不一致，则下载新文件并保存在缓存内。它的作用与 `Cache-Control: no-cache` 或 `max-age=0` 相同。

对于经常过期的文件，`ETag` 是验证本地缓存文件是否依旧与服务器同步的好方法。但对于很少改动的文件，`ETag` 则是昂贵（从性能方面来看）的缓存机制。这是因为尽管并未下载文件（这样节省了带宽），但是依旧建立了连接，连接时间延长了文件处理过程。

下面例子中有一个 `ETag` 报头和一个 `Cache-Control` 报头。在 86 400 秒（1 天）内，设备会从缓存读取文件，之后检查 `ETag`（或最后修改）报头以查看文件是否被改动。如果无改动，则继续使用缓存 86 400 秒：

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=86400
Content-Type: image/jpeg
Date: Tue, 28 Jan 2014 00:14:55 GMT
Etag: "b17ad00-1f17-46723595372c0"
Expires: Wed, 29 Jan 2014 00:14:55 GMT
Last-Modified: Thu, 09 Apr 2009 18:23:47 GMT
```

```
Server: Apache/2.2.3 (CentOS)
X-Cache: HIT
Content-Length: 7959
```

5. Expires

Expires 报头不如 Cache-Control 和 ETag 报头常见（但同样有效），它并不给出按秒计算的文件过期时间，而是给出文件未来过期和应当重新验证的具体日期。这是网络上最早使用的缓存报头，一些老式的浏览器可能还在使用它。Expires 报头应与 Cache-Cotrol: max-age 一致。在前例中，Expires 报头恰好是从服务器获得文件的一天后。

比缓存更快吗

你的 App 会在第一次启动时下载内容并缓存很长时间吗？记住：第一次启动几乎决定了用户满意度，关乎成败。如果 App 第一次启动花费了很长时间进行配置（下载图片和文件），那么用户可能就不会再用你的 App 了。把图片和文件放入 App 的资源文件，虽然需要下载的 App 会变大，但是可以加快第一次启动的速度。此外，如果你改动了标志、图标等，你需要做的是发布一个 App 更新。

你可以使用 ARO 查看 App 是否正确缓存。有三个最佳实践可以帮助你确定缓存文件的问题：重复内容、缓存控制和内容过期。图 7-7 中的表格显示了一个跟踪里下载超过一次的文件列表、每个文件的下载次数和重复文件的大小。ARO 里的缓存控制（Cache Control）和内容过期（Content Expiration）最佳实践充当的是服务器端和设备端（各自）潜在缓存问题的警告。

ARO 缓存控制最佳实践用于查找是否存在 Cache-Control/ETag 或 Expires 报头。如果服务器未插入此类报头，则发出“此处缓存策略可能失败！”的警告。可能你不想缓存某个文件，所以就省去了报头。但是这里有一个重要问题需要注意：HTTP 缓存规范规定，如果文件不含内容，那么会被缓存 24 小时。如果你不想缓存文件，必须明确说明，以避免出现按照规范进行缓存的情况！

ARO 内容过期最佳实践确保了 App 缓存正常工作。它计算“304 not modified”服务器检查消息的数量、缓存报头被忽略的次数以及向服务器请求文件的次数（当文件应当在缓存中时）。通常来说，它会向设备上未配置（或未正确配置）缓存的 App 发出警告。

如果 App 重复下载内容，你就要多加注意，确保正确插入报头（服务器修复），而且 App 在缓存里正确地保存文件（应用修复）。

7.3.5 文件之外

优化下载的文件很重要。更小、更精的文件常常可以使下载更快速，性能更高效。但是，

现在你也知道蜂窝网络延迟和 7.1.3 节中的 RRC 状态机是如何影响下载速度和用电量的了。假设所有文件都已经被优化，现在我们需要确保连接 App 和服务器以获取这些文件的过程要尽可能地高效，并与 RRC 状态机协作，从而最大化性能和用户满意度。

7.3.6 分组连接

想象一个广告支持的图片分享 App。我们直觉地知道，传输图片的连接会占用很大带宽，并且会经常连接网络。你知道广告 SDK 的行为吗？广告是和图片一起载入的吗？当无线电静默的时候也会发生这些连接吗？你的分析数据又是怎样的？这些连接是同时发生的吗？这些连接在任何时候都可以随心所欲地唤醒无线吗？

你可能会认为，构建这些工具就是为了连接。如果除了库和连接外，你还使用了多个其他的分析提供程序（或广告服务），那么 App 就永远也不会让无线电休眠了，因为所有的服务想什么时候连接就什么时候连接。如果想让 App 尽可能高效，将连接编组为几个大的 bucket（而不是很多个小的 bucket）是可行的。查看这些 SDK 的文档和代码，看看是否可以将它们与 App 里的其他连接同步。如果测试发现第三方 SDK 的表现达不到其应有的水平，联系它的开发者。开发者可能也不清楚库的表现，并且可能会有兴趣改善库的网络表现。

定期连接

由于 RRC 状态机，每个服务器连接都会使无线电保持在活动状态，所以尽可能减少 App 内的（特别是后台发生的）连接数量非常重要，这样不仅能延长电池寿命，也能减少用户的流量。2013 年，我的团队与一个流行社交媒体 App 合作，目的是减少 Android App 在后台产生的连接数量。在这个 App 的早期版本里，我们看到有三个连接以不协调的方式在后台运行。每 30 分钟，这三个连接打开无线电七次。在图 7-13 中，这 30 分钟以红色突发（Burst）的数据包为界线。在图片上部，你可以看到两个发生时间非常接近（但不重叠）的紫色、黄色和蓝色突发。紫色连接打开第二个和第三个连接，黄色突发再次使用这两个连接，蓝色突发是服务器发出的数据包，提醒 App 关闭连接。

开发人员看到这个后就知道，简单地协调这些连接（并确保正确关闭它们）可以大大减少 App 的后台电量消耗。下部的跟踪显示了改进情况。App 的“升级版”将这三个连接整合到一个事务时间，刷新率被开发人员加倍至每 15 分钟。现在，每 30 分钟连接 2 次，数据更新频率是原来的两倍，而用电量却下降了不止 50%。假设这些连接一天 24 小时都发生，估计我们可以为每个安装了此 App 的用户实际节省约 5% 的电池电量。

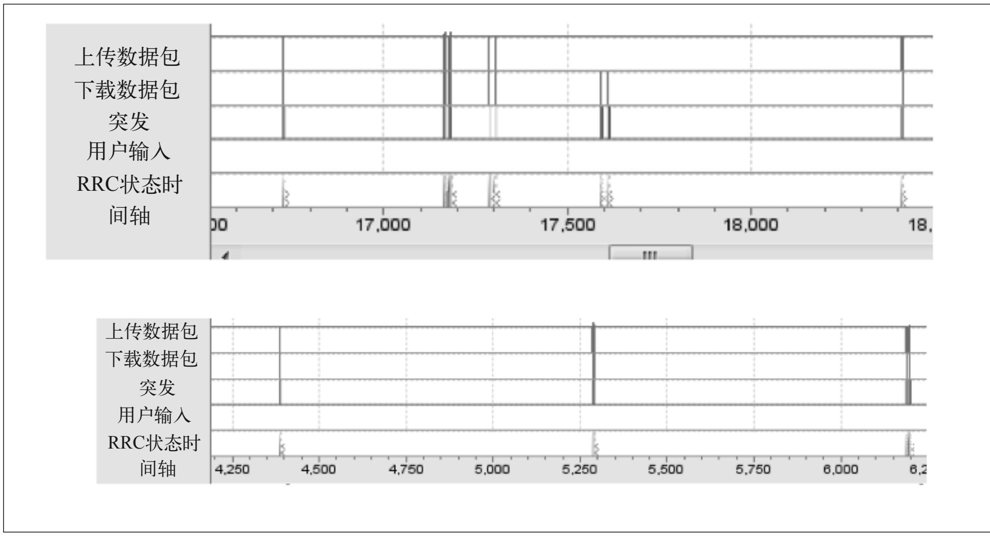


图 7-13：社交媒体后台连接：优化前（上）和优化后（下）（另见彩插）

并不是所有开发人员都有时间制作自己的事务管理器，但是 Android 开发人员一直都在注意这个问题。在 3.6 节中，我们讨论了 Android 5.0 引入的 JobScheduler API，并举例说明了如何让操作系统减少周期性连接，从原先可能多达 20 个的连接减少为 9 个。将 JobScheduler API 的灵活性加入到下载非关键性元素中，并在后台连接加入回退机制，这可以大大降低无线电的使用率，并能在提高 App 性能的同时，降低用电量（你和用户的双赢）！

7.3.7　检测应用的无线电使用情况

要确定用户设备连接的是 Wi-Fi 还是蜂窝网络，你可以查询连接管理器，如例 7-1 所示。

例 7-1：连接识别：Wi-Fi 或蜂窝网络

```
public static String getNetworkClass(Context context) {
    ConnectivityManager cm = (ConnectivityManager)
                             context.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo info = cm.getActiveNetworkInfo();
    if(info==null || !info.isConnected())
        return "-"; //未连接
    if(info.getType() == ConnectivityManager.TYPE_WIFI)
        return "wifi";
    if(info.getType() == ConnectivityManager.TYPE_MOBILE){
         return "cellular";
         }
     }
     return "unknown";
}
```

通过这个数据片段，你可以知道使用的是哪种连接，然后针对这两种网络类型定制数据流。如果用户使用的是蜂窝网络，可以使用能推迟传输的非紧急通信。在 Android 5.0 的 JobScheduler 出现前，我们无法得知网络的使用情况。使用例 7-2 中所示的代码可以让分析和广告只在蜂窝网络已使用的情况下载入。

例 7-2：确定蜂窝网络连接是否存在

```
if (Tel.getDataActivity() >0){

          if (Tel.getDataActivity() <4){

          //1，2，3响应代表蜂窝网络正在传输!
          //用image getter下载图片
           imagegetter(counter, numberofimages);

           //并显示广告
           AdRequest adRequest = new AdRequest();
           adRequest.addTestDevice(AdRequest.TEST_EMULATOR);
           adView.loadAd(adRequest);
           //发起通用请求以便随广告加载
           adView.loadAd(new AdRequest());
}}
```

这个代码片段使用 `TelephonyManager`（`Tel`）数据行为 API 确定无线电是否打开。如果无线已经打开，则使用这个连接下载更多内容。它只表明数据传输是否在蜂窝网络（不是 Wi-Fi）上发生的。在 Android 5.0 中，`ConncetivityManager` 中加入了新 API，现在可以利用 `ConnectivityManager.OnNetworkActiveListener` 将这个仅适用蜂窝网络的方法推广应用到所有无线连接，从而找出无线网络何时处于高功率状态（并准备传输数据）。可以使用 `ConnectivityManager.isDefaultNetworkActive()` 查看网络是否已经处于活动状态。使用已经建立的无线连接是共享资源和节省用户电量的好方法。

GCM Network Manager

在 2015 年的 Google 开发者大会上，Google 和 Android 使省电的网络连接调度更为简单。作为 Google Play 服务的一部分，它们模仿 JobScheduler API 连接添加了 GCM Network Manager API。但是 JobScheduler 只能在使用 Lollipop 的设备上运行，而 GCM Network Manager 可以在 Gingerbread（2.3）版本之后的所有 Google Android 设备上运行。 现在，和在 JobScheduler 中一样，你可以简单地设置连接，只在 Wi-Fi 开启状态或设备连通电源的时候运行。你也可以设置任务使其在后台定期运行或自动退出。利用此 API 进行非紧急性更新与连接，可以为用户直接节省大量的设备电池用量。

7.3.8 适时关闭连接

如果在手机上建立无线电和 TCP 连接时发生延迟，你可能会认为明智的做法是只对服务器开启 TCP 连接。 这样一来，如果更多的数据包需要发送到设备，你就可以减少一些连接

设置延迟。对相对快速、连续发送的文件确实如此。但如果文件发送的间隔时间达到或超过了 15 秒，可能还是得开启无线电（其实连接设置所需时间极少）。

如果一段时间内没有产生数据流量而连接还在开启，设备和服务器都有个关闭连接的清理进程。这也不是件坏事（后续的章节会讲述原由）。但缺点就是关闭连接的一方会告诉对方：“喂，我现在要关闭连接了。”这可能会造成无线电连接的继续开启，继续在设备上运行 10~15 秒的 RRC 状态机，给用户造成额外的电量损耗。

从图 7-14 中 ARO 展示的截屏来看，在第 8 秒的时候一个小图片被加载了，但是连接并没有断开。

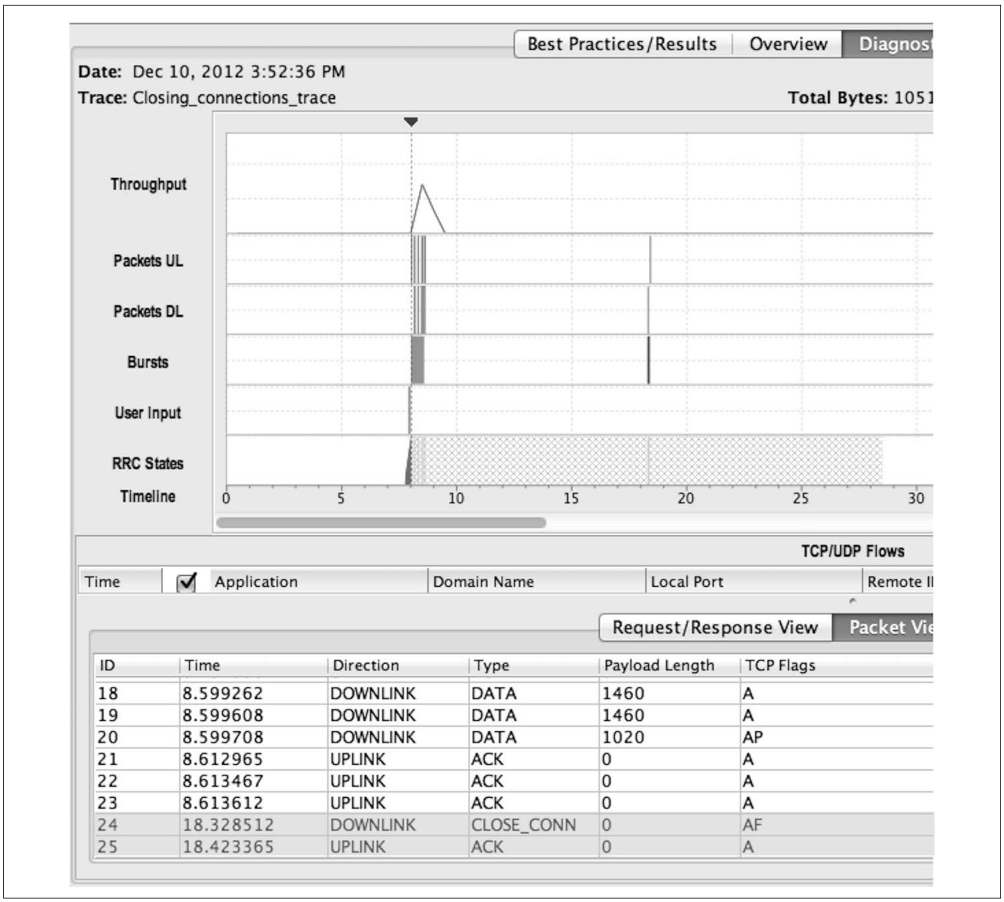


图 7-14：ARO 诊断选项卡显示连接关闭问题

在第 18 秒的时候，服务器（很可能在执行清理进程）关闭了连接，造成 RRC 定时器重置（在数据包查看表中：18 秒时，服务器数据 ID24 关闭了连接）。但无线电连接并没有在 18~19 秒的时候关闭，而是持续开启到 28 秒，这几乎耗费了下载一张图片的两倍电量。

对于那些不再需要的连接，可以在下载结束时指定关闭。例 7-3 中，我禁用了“保持连接”。然后下载结束时，我就断开了连接。这就表明连接资源可以重复使用或者关闭（节约内存等）。

例 7-3：适时关闭连接

```
HttpURLConnection connectionCloseProperly = (HttpURLConnection)ulrn.openConnection();
//这禁用了“保持链接”
connectionCloseProperly.setRequestProperty("connection", "close");
connectionCloseProperly.setUseCaches(true);
connectionCloseProperly.connect();
Object response = connectionCloseProperly.getContent();

InputStream isclose = connectionCloseProperly.getInputStream();

        ...download and render bitmap image

connectionCloseProperly.disconnect();
```

一旦执行这段代码，图片下载完成后，服务器和设备就会立即关闭连接。

7.3.9 定期执行重复的ping命令

那些需要定期更新数据的 App，应该使用 Google Cloud Messenger 之类的工具向 App 推送更新信息。你可以定制个性化服务，通过设置提醒每 *x* 分钟后开始后台运行，唤醒无线电连接并下载数据。这似乎不是什么大事，但是想象一下，如果 App 每三分钟就 ping 一次服务器以获取更新信息，那么 App 每 24 小时就要连接 480 次。置入一个 10 秒的状态机定时器，这些“无害”连接每天要耗费 80 分钟进行无线电连接。如果必须定期唤醒无线电连接获取数据，必须确保可执行回滚操作，或在特定时间段后禁用提醒以保持 App（或设备）休眠。

在一些情况下，App（如实时游戏）可能需要保持连接上的数据包交换。在这种情况下，务必将发送的数据量减到最少，同时要明白 App 运行时保持无线电连接开启是非常耗电的。

完美风暴：重复连接和关闭连接

想象一下，当几个人在某区域移动，他们手机上的 App 每隔五秒和服务器交换一次实时数据以更新位置信息。再想象一下，收到最后一个数据包后（在服务器上保留 IP 地址），服务器上的连接仍然保持 90 秒。通常而言，如果每个用户每次会话只要占用一个连接，这就应该没什么问题了。但是如果你更改了代码中的配置，使每一次 ping 都与服务器建立一个新的 TCP 连接，而 App 在发布前没有测试到这个问题，最后会发生什么事？

那么你就引发了一场完美的数据流量风暴！现在每个 Android 用户每 5 秒 ping 一次，将 18 个之多的 IP 地址连入服务器。连入的用户越来越多，可能就会发生 IP 冲突现象，从而导致用户无法连接！恭喜，你的 App 向服务器成功地完成了一次 DDoS 攻击。

因此，要特别注意服务器重复执行 ping 操作，并且应当在 App 发布前进行测试。

7.3.10 网络安全技术的应用（HTTP和HTTPS）

在用网络传输数据时，必须保证用户的个人数据安全。似乎每周都会爆出一款移动 App 存在严重的个人信息安全漏洞。在本地设备和服务器上恰当存储文件非常关键，但是在网络上正确传送文件也很重要。用户可能会连接到任何类型的网络，包括服务场所不安全的 Wi-Fi 热点。如果你传输的数据通过 HTTP 发送，窥探者能毫不费力地获取该数据——因为你是用明文发送的！你应该通过 HTTPS 发送，使用密钥加密数据，然后再将密钥告知接收方。虽然初始化连接会造成额外的数据往返，但是只要正确配置了 HTTPS 连接，这个传输过程就是安全的。

7.4 全球移动网络覆盖范围

关于好房子，地产中介有句口头禅："位置、位置、位置。"如果按照前文所说的那些最佳实践彻底优化网络，那么你在移动网络性能上就迈出了漂亮的第一步。然而，尚有一项最重要的变量我们没有考虑到——用户的网络速度。（很显然）我们不能控制用户以什么方式或在什么地方连接到我们的 App。不过，我们可以尽量地优化用户体验。

GSMA 机构的信息显示，2014 年，近 40% 的移动网络连接设备是智能手机（至 2020 年，此比例会上升至 65%）。

如图 7-15 所示的全球移动网络市场渗透率，2014 年第三季度，4G 的市场渗透率大约是 5%，3G 略微超过 30%。这意味着，有相当大一部分用户（至少 5%，如果我们只考虑不具备 3G 功能的手机）使用专用 2G 网络的智能手机。

2013 年，百度宣称中国有 2.7 亿的 Android 用户，而其中 31%的用户使用 2G 网络连接。之后，LTE 在中国推出，但我们还是很明显地看到，很多 Android 用户仍然只使用 2G 网络进行数据连接。

智能手机使用慢速网络并不只是美国和欧洲以外地区才存在的问题。即使在发达国家，有些地方也并没有开始推行 LTE（或高速网络只有零星覆盖）。用户将来可能去到这些地方并使用你的 App，因此，有必要确保移动 App 在慢速和更拥挤的网络中运行良好。

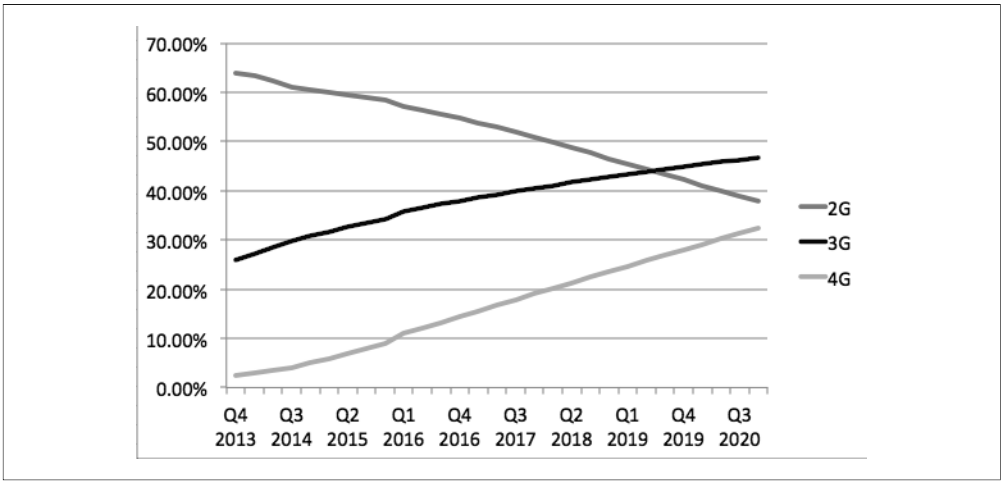


图 7-15：全球“G”市场占比（GSMA 机构供图）

在 2.4 节中，我们探讨了为何“你的设备不是用户的设备”，同样，你的移动网络不是用户的移动网络。大部分的开发者处在高速网络覆盖的区域，至少是 3G（很有可能是 4G LTE）无线电连接。开发者除了拥有大屏幕和快速的处理器之外，还生活在高可用性网络的良好环境中。对于和开发者有相同环境的用户来说，这当然很好。然而，了解世界各地的网络连接才可以确保我们的确在为最终用户提供正确的数据。

7.4.1　CDN服务器

由于延迟是移动数据通信的主要绊脚石，任何能减少延迟的事情对最终用户来说都将加速数据的获取和 App 的渲染。虽然光的速度快得令人难以置信，但它也需要花 53 毫秒从波士顿往返于伦敦（从波士顿到悉尼需要 162 毫秒）！为了减少延迟，应当考虑使用 CDN 在世界各地的数据中心制作内容的镜像，从而让用户可以更快地获取到他们需要的数据。

CDN 服务器（大体上来说）是用来在网络前沿或者接近最后一英里处存储数据的服务器。它依赖于数据存储的分布式系统，主系统不会因为请求次数太多而不堪重负。将这些 CDN 服务器放置在用户附近，数据就更靠近用户了，这样就减少了请求和传送文件的往返时间。

Facebook 报告显示，在印度尼西亚，50%的手机用户使用 Facebook，而其中 75%的用户使用 2G 网络。在连接有限的情况下，面对大量用户，Facebook 尝试尽可能实现收益最大化。印度尼西亚是一个幅员辽阔的大群岛，而该国大部分地区距离新加坡（一个可能的 CDN 位置）3200 公里。数据从本地 CDN 服务器到用户手机在光纤中的往返时间大约需要 32 毫秒（假设光在光纤中的速度为 200 000 千米 / 秒）。即使有这样大的延迟，Facebook 发现还是需要积极地进行 CDN 映射。通过分析，Facebook 发现，仅 16% 的流量来自本地

CDN 服务器，而 84% 的图像或者视频来自南美（跨越半个地球）。对于从南美传输到印度尼西亚的数据来说，它必须要跨越太平洋（http://www.submarinecablemap.com/）。哪怕我们将 CDN 服务器放置在厄瓜多尔（南美洲的最西端），数据仍然必须传输大约 18 000 公里，往返时间为 176 毫秒（是新加坡 CDN 服务器的 5.5 倍）。这无疑显示了 CDN 的价值。此外，精心调节 CDN 流量以缩短数据传送给用户的距离也很重要。

7.4.2 在慢速网络中测试App

测试 App 在慢速网络中的表现，第一步应该是申请一次环球旅行，考察 App 会被用到的地方（这没有什么大不了的，对吧？）。从 7.4.1 节可知，Facebook 已经在非洲和印度尼西亚进行了测试，并且公布了一些很有趣的结果。通过在非洲一个月的数据统计，Facebook 发现其 App 在 40 分钟内会崩溃。这次统计促使 Facebook 竭尽所能地减少数据使用量和网络利用率，并优化了图像，做更好的缓存。最后 Facebook 的数据使用量减少了一半（获得了所有用户的一致好评！）。

很少有公司具备和 Facebook 一样的资源条件，因此无法通过环球旅行测试 App 是可以理解的。然而，仔细分析调研数据，挖掘各地区、各国家的延迟时间和带宽问题，还是可以得到一些有用信息的。

7.4.3 仿真慢速网络而不用倾家荡产

在第 2 章中，我建议使用专用的 Wi-Fi 网络进行数据测试。如果设备实验室只在高速网络下进行所有的测试，那么就有可能漏掉慢速网络下重要的测试情景 。如果不考虑移动网络吞吐量的差异性，将难以吸引新用户，还会降低那些到差网络环境下的现有用户的使用积极性。运营商会在一个封闭的环境中使用专门的天线测试各种情况。但是搭建这些环境是非常昂贵的，那么怎样才能很好地进行测试而又不用倾家荡产呢？让我们一起看看以下这些方法吧（按实现的成本列出）。

1. Wi-Fi网络节流

如果你正在使用无线路由器进行测试，并且可以在路由器上安装 OpenWRT（一个开源路由器），那么会有一个 wshaper 插件（http://wiki.openwrt.org/doc/recipes/guest-wlan），让你可以对上行和下行链路连接进行节制，这至少可以让你模拟较慢的网络速度（但不是延迟）。

2. 模拟器

Android 模拟器可以控制网络条件。打开模拟器后，你可以登录到模拟器以模拟不同的吞吐量和延迟：

```
telnet localhost 5554
network speed edge  //gprs, umts hsdpa和全面的附加选项
network delay edge
```

3. 自制法拉第笼

法拉第笼是由金属丝构成的笼子，它将内部空间同外部的电磁辐射完全隔离开来。制作局部法拉第笼能够减少到达手机的信号量。有报道称，一些开发人员成功使用旧的（不插电！）微波炉屏蔽掉了部分无线电。由于实验的不确定性，这些测试的结果很难重现，但这也许已足以用于定性测试了。

4. 网络弱化器

AT&T 发布了一款名为 AT&T 网络弱化器（http://developer.att.com/attenuator）的工具，如图 7-16 所示。这款网络弱化器在 Samsung S3 ICS 内核上运行（需要 root 和一个由 AT&T 提供的自制 ROM 闪光）。安装后，这个 App 可以像一个刻度盘一样减慢移动网络、降低吞吐量（很抱歉，如果你连接的是 3G 网络，它并不能使你的连接变得和 4G 网络一样快）。当滑动滑块将网速从 UMTS 变为 EDGE 时，上行链路、下行链路和往返时间定时器都会相应地作出调整，让你在较慢速的网络条件下对 App 进行一些简单的测试。你也可以从左到右滑动滑块以调整网络拥塞程度，增加每个连接的往返时间。

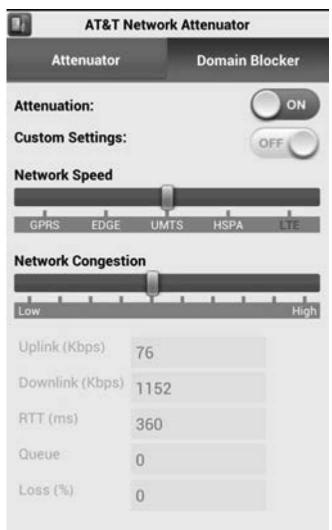


图 7-16：网络弱化器 APK

7.4.4 构建网络感知App

如果你知道用户将会连接到不理想的网络（而且我们知道他们肯定会），那么保证他们的最佳用户体验难道不是理所应当的吗？我不是建议降低 App 在 3G 或者 4G 网络下使用的质量，而是有技巧地加强 App 在慢速网络下的体验。我喜欢称运用了这种架构的 App 为"灵活的网络感知"（FNA）App，因为它们可以感知网络，并且根据所测得的网络条件，调整用户的体验。下面一起来看一下"网络活动示例"App（https://github.com/dougsillars/

FNASampleApps/tree/master/NASampleApps/NetworkActivitySample）的代码。在描述 App 时大声地说出运用了 FNA 技术是一件很酷的事情。

要为连接快速、中速和慢速网络的设备提供不一样的移动体验 ，只需要简单地去除内嵌视频或者减少图像的数量（或者至少改变图像的大小）。例 7-1 的代码是用来进行移动网络连接的，你可以查看 `TelephonyManager` 类。像例 7-4 一样，你可以使用 App 转变展示体验的类型 。

例 7-4：确定移动网络速度

```
TelephonyManager teleMan =
   (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
int networkType = teleMan.getNetworkType();
switch (networkType)
{case 1:    netType = "GPRS";
                        networkSpeed = "slow";
                  break;
case 2:     netType = "EDGE";
                networkSpeed = "slow";
                        break;
case 3:     netType = "UMTS";
                        networkSpeed = "medium";
                    break;

// 我们会略过一些网络类型,但你应该明白我的意思了
// 你可以在GitHub中查看完整代码

case 13:    netType = "LTE";
                        networkSpeed = "fast";
                        break;}
```

通过对网络状态的定期检查，FNA App 将会根据当前的网络情况适当地增强或者减弱性能。这是一个非常简单的算法，并且不需要考虑网络的强度。进一步而言，你可以认为弱的 3G 网络是慢速网络，或者弱的 4G 网络是中速网络。

例如，你可以用 App 在慢网上下载一张小图片，在中速网络上下载一张中等大小的图片，而在快网上下载一张大图。和上文类似，你可以认为 Wi-Fi 是快速网络，甚至是“高速”网络，只是通过 Wi-Fi 向用户发送数据并不会产生流量费用罢了。

例 7-5：设定网速

```
switch(networkSpeed){
case "fast":
        new ImageDownloader().execute(urlbig); //图片大小为 143KB
        break;
case "medium":
        new ImageDownloader().execute(urlmed); //图片大小为 41KB
        break;
case "slow":
        new ImageDownloader().execute(urlsmall); //图片大小为 27KB
        break;
}
```

下载图片的时候，我计算了下载的实际耗时，为了计算更准确，我记录了两次：从发送请求到收到来自服务器的 200 回复的时间，以及下载图片的总用时。响应时间（收到服务器的 200 回复的时间）是往返时间的两倍（这里假设 DNS 查找已提前完成），它可以用来估算网络延迟。下载时间是从服务器接收到对象的总用时。另外，通过查询内容的长度，Android App 能够计算出该文件以 KB/ 秒为单位的实际吞吐量。

例 7-6：确定实时的往返时间和吞吐量

```
private Bitmap downloadBitmap(String url) {
   Long start = System.currentTimeMillis();                //下载开始时间
       final DefaultHttpClient client = new DefaultHttpClient();
 final HttpGet getRequest = new HttpGet(url);
       try {HttpResponse response = client.execute(getRequest);
           //检查成功返回码200
           final int statusCode = response.getStatusLine().getStatusCode();
           //收到200成功码回复的时间
           Long gotresponse = System.currentTimeMillis();
            }
           final HttpEntity entity = response.getEntity();
           //获得文件的ContentLength
                       contentlength = entity.getContentLength();
           if (entity != null) {
              InputStream inputStream = null;
              try {
  inputStream = entity.getContent();
  final Bitmap bitmap = BitmapFactory.decodeStream(inputStream);
                Long gotimage = System.currentTimeMillis();
                               //图片下载完成的时间
                responsetime  = gotresponse - start;
                               //200回复完成的时间
                imagetime     = gotimage-start;
                                                       //下载时间
throughput   = ((double)contentlength/1024)/((double)imagetime/1000);  //KB/s
                return bitmap;
             }
```

那么，这些数据告诉了我们什么呢？用网络弱化器 App 模拟各种不同网络速度，能够计算出上述三种网络环境下三种不同大小文件的下载时间，结果如表 7-2 所示。

表7-2：图片下载时间

文件	LTE	UMTS	EDGE
大文件 (143 KB)	1.938	5.243	9.405
中文件 (41 KB)		2.793	
小文件 (27 KB)			3.401

如果所有网络条件下都使用大文件（非 FNA App），那么很显然，由于文件较大，使用 UTMS 和 EDGE 的用户体验会明显较慢。如果运用 FNA 框架，UMTS 的下载速度会增加将近一倍，而 EDGE 的下载速度会增加将近三倍。然而不可否认的是，尽管这个简单的模

型显示了它在改善用户交互方面的优势，但使用网络技术判断下载速度从而估算理想网络速度是一种非常粗糙的方式。

在顶级的网络环境中收集延迟和吞吐量的数据能帮你构建一个更好的算法，使 App 可以根据接近实时的网络环境灵活地进行调整，但显然简单的算法对用户更有益。

测量延迟

据我所知，测量 RTT（往返时间）有很多的可变因素。到蜂窝基站的距离、拥塞，或者是来自其他无线电的干扰都能引起 RTT 的剧变。因此，不能仅依赖于一两个离散的测量值，而应该先去掉任何潜在的异常值后取测量的平均值。虽然根据当前的网络条件进行运作是很棒的，但平稳地输出数据也是至关重要的。

7.4.5 计算延迟

如果你的 FNA App 感知到用户正处于一个高延迟的环境当中（根据计算得到的高 RTT），它能够通过更积极地预加载帮助用户加速体验。例如，如果用户正在浏览一系列的图片，它可以在用户到达列表尾端之前先下载其他图片（例如，屏幕上只有两张图片，开始下载下一屏图片）：

```
if (ImagesBelowtheFold<2){
                <get next batch of images>
                }
}
```

在高延迟的环境中，用户可能在下一屏的图片加载之前就滑到了列表的末尾。考虑到这种情况，你可以提前获取下一屏的图片：

```
If (latency = normal){
        if (ImagesBelowtheFold<2){
                <get next batch of images>
        }
}
Else {
        //延迟严重
        if (ImagesBelowtheFold<4){
               <get next batch of images>
       //考虑获取更多图片
       //同时,更小的图片?
        }
```

将开始下载的时间提前两倍，相当于在用户注意到延迟之前给网络两倍的时间（相较于之前）来下载数据。 这样可能占用网络的时间稍长，并下载更多的图片（使用更多的数据），所以它需要结合情境使用。但是如果能让用户得到无缝体验，这样做也许是值得的。

7.4.6 最后一英里的延迟

延迟通常在数据传输的最后一英里发生，在移动网络中尤其如此。这些技巧可以帮助你应对延迟，缓解问题，但不能从根本上解决问题。 正如在 7.4.2 节中描述的那样，Facebook 发现，在印度尼西亚的慢网连接中，84%的流量来自于南美和欧洲的 CDN 服务器。

7.4.7 其他无线电

Wi-Fi 和蜂窝无线网都是最常用的传输数据的网络，也是最容易优化的。但是其他的无线电网络也会导致设备的能耗，因此它们的行为也应该被关注、讨论。

7.4.8 GPS

Android 提供了模糊定位，使用附近蜂窝基站和 Wi-Fi 网点的信息即可生成粗略的位置信息，并不需要开启 GPS。然而，很多 App 需要更精准的定位，它们会打开 GPS 设备，从 GPS 卫星接收信号。定位需要从手机到卫星之间的一条信号线。

为了优化定位设备的使用性能，可能必须调整 GPS 打开的时间（保持 GPS 接收器打开多久），以及使用频率。打开的时间越长，使用的频率越高，你得到的位置信息就越精准。

7.4.9 蓝牙

目前，所有的 Android 旧设备都必须通过蓝牙才能连接到其他设备。如果你对蓝牙的数据传输感兴趣的话，可以在 Wireshark 的非集群中收集它的日志信息。对于使用 KitKat 和更新版本系统的设备，你可以在开发者选项设置中打开“Bluetooth HCI snoop log”。当你选中了该选项时，Android 设备将会收集所有通过蓝牙接口发送的数据包的日志信息，信息会被存放在 /sdcard/btsnoop_hci.log 中。

在 Wireshark 中打开这个日志文件，你可以看到被传输数据包的信息。大部分数据是加密的，但是你可以观察到两个设备之间的通信模式（见图 7-17）。

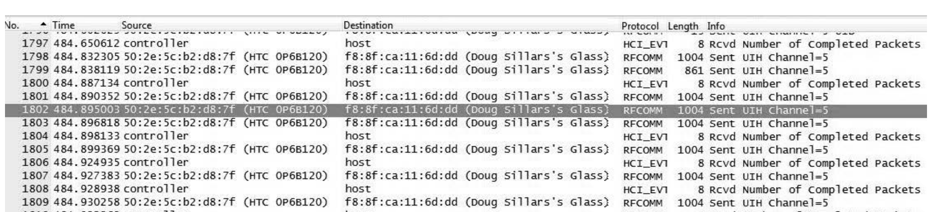


图 7-17：Wireshark 中的蓝牙通信

在这个例子中，请求的响应是一个 POST，你可以查看我在谷歌查询“澳大利亚牧羊犬”（来自 Glass）的响应信息（见图 7-18）。

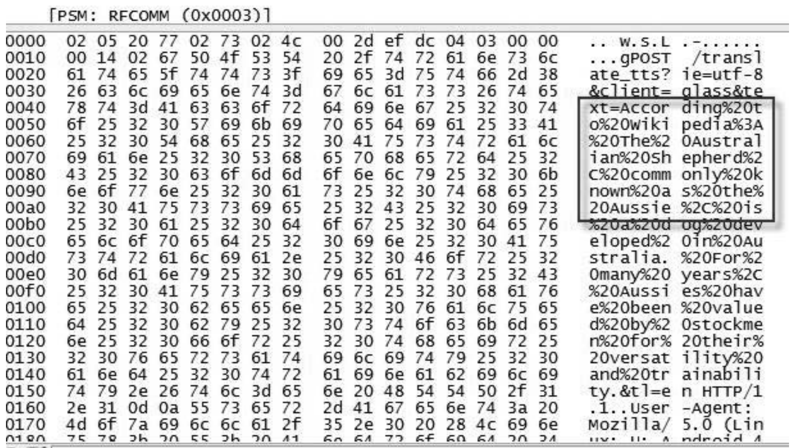


图 7-18：蓝牙 POST 响应

使用 Wireshark 工具，可以看到一段时间内蓝牙传输数据包的数量（见图 7-19）。

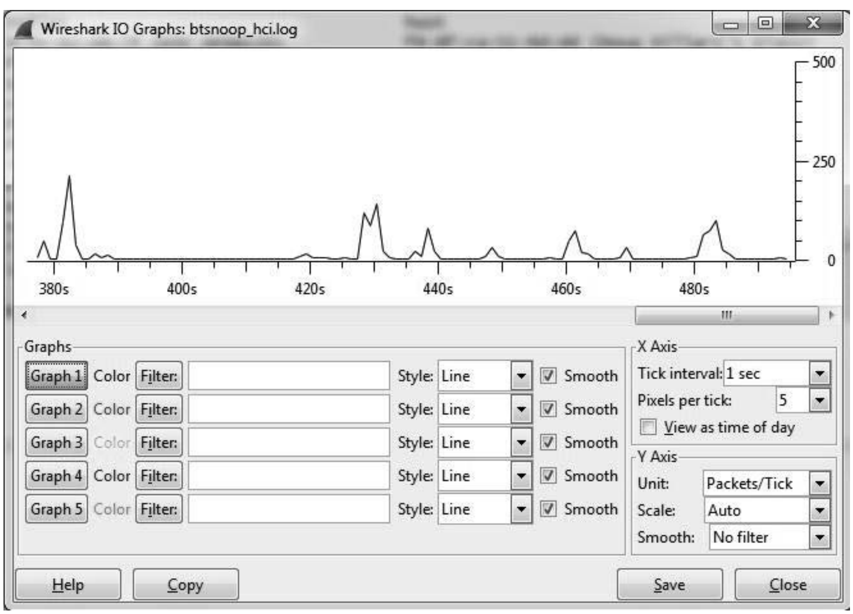


图 7-19：蓝牙 POST 响应 – 包的数量随时间的变化

7.5 小结

大多数的 Android App 使用蜂窝和 Wi-Fi 无线网络与外部服务器进行数据通信。无线电设备是仅次于屏幕的第二大耗电设备，因此使用它们的时候需要非常谨慎。此外，大多数用户每个月的流量是有限的，那么在移动网络传输文件时，在速度和流量消耗上选择更优质的策略则显得更为重要了。除了考虑数据流量的成本之外，也必须考虑高延迟和较慢的网络速度。无论用户在哪，无论当前的网络环境怎样，确保数据传输对当前的网络环境来说是优质的，将会使你的 App 脱颖而出。通过 RRC 状态机优化数据流量和用户的位置信息，将会延长手机电池的使用寿命并且改善所有用户的体验。

第 8 章

真实用户监测

前面章节介绍了一些优秀的工具来诊断 Android App 中的问题和缺陷。我们研究了一些可以在 Android 设备上使用的免费工具，用来优化电池、内存、CPU 和网络。然而正如大家所知，我们在第 2 章中也讨论过，这些测试需要能连接互联网的物理设备。

如果只有很少的差旅预算和有限的设备预算（以及无限的时间来专注于性能），你怎么确保不管在什么位置、设备和网络条件下，App 都能保持最佳性能呢？答案是收集 App 的运行时数据，统计结果，生成报告，从这些数据中寻找可能出现的问题。这些从 App 自身中得到的分析，就是众所周知的真实用户监测（RUM）。

虽然有些开发团队拥有足够的资金，可以建立自己的 RUM 引擎来收集数据，但其实市场上已经有很多工具可以集成到 App 中，它们可以从安装的设备中收集数据。这些工具中很多是免费的或限免的，允许你收集信息而不必付出巨大的前期成本。如果 App 开始收集大量的数据或者你需要详细的报告，你必须为这些服务付费，但是这些数据的价值（就像你看到的）是值得你为之付出的。

不仅仅针对大型团队

RUM 数据收集不仅存在于拥有专业性能优化团队的大公司中。也许你自己就是一个性能团队（同时还承担着各种其他任务）。在 App 中收集用户的数据并不难，并能帮助你认识到如何提高 App 的可用性和性能。你一定要试试，我敢保证这简单的前期工作必将让你受益匪浅。

8.1 启用RUM工具

市场上有许多可用的 RUM 工具，每个工具都有一些可能用得着的略微不同的报告和数据。为了收集各方面的完整信息，可能需要在 App 中安装多个 SDK。每个 RUM 工具都提供了将代码库集成到 App 的详细说明，一些工具甚至集成为一个自动化的安装程序，这样就不需要额外的工作了。大多数功能齐全的 SDK 可以帮助建立指标以监测和收集数据。在这一章，我选择了三个 RUM SDK 添加到示例 App（Image Scroll）中，由此来看看我们可以获得什么数据。

要加入的第一个 SDK 是 Crashlytics，它是一个非常简单的插件（如图 8-1 所示）。点击安装按钮后，代码将自动被添加到 App 上以支持这些分析。

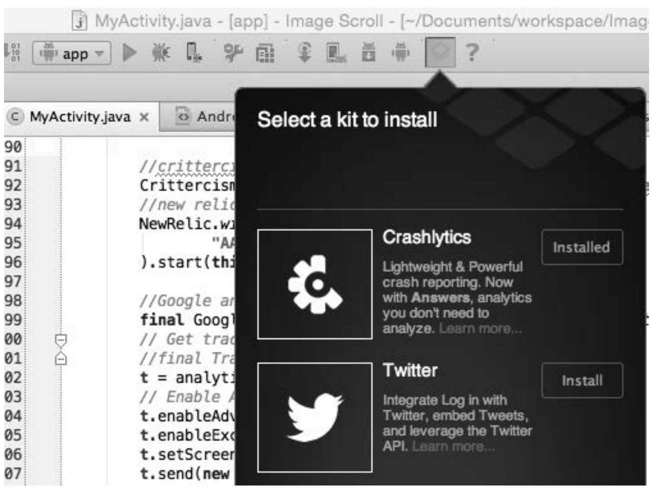


图 8-1：安装 Crashlytics

安装 SDK 后，只需要创建和分发 App。随着客户开始使用 App，使用情况会被统计成报告，反馈给 RUM 供应商。Web 信息显示板可以用来研究用户使用 App 的情况，并发现 App 中存在安全隐患的地方。通过远程识别这些问题，你将不必再依赖来自于用户的 bug 报告。你可以直接修改这些 bug，并在下一次 App 升级时发布这些补丁。这样可以确保 App 在所有设备上以最佳状态运行。

8.2 RUM分析：示例程序

这些工具是如何收集用户信息的呢？通过引入 JAR 或库（有时候仅仅是一些代码），这些 RUM 工具在 App 运行的时候就会开始收集数据，然后将这些数据上传到会产生数据信息板的服务器上。当 App 发生严重错误时，这些服务器就会报警。

每个工具都有不同的地方，但是大多数工具都可以将数据按照区域、设备、操作系统、App 版本或其他标准划分。为了获取示例的 RUM 数据，我建立了一个叫作“Image Scroll”的 App。当手指滑动时，它可以一次加载 10 张图片（见图 8-2）。这些图片托管在一个远程服务器上，而图片的 URL 则存储在一个数组中。当 App 达到数组的边界（在这里是 92 张图片）时，Android 会抛出一个 out-of-bounds 异常并导致 App 崩溃。这样的设计可以通过设备跟踪 App 的崩溃。

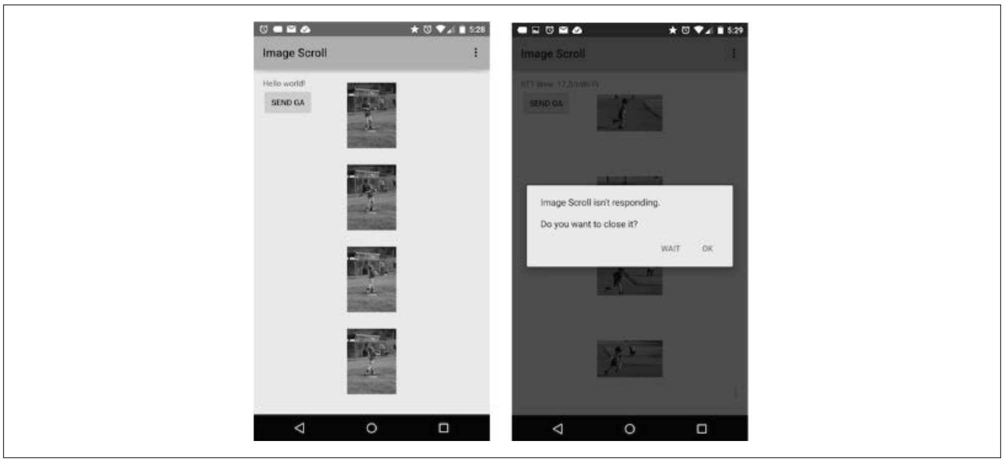


图 8-2：Image Scroll App

此 App 有几个错误的图片 URL（为了产生 404 错误），并在第二组中将一张大小约 50KB 的棒球运动员图片替换为一张大小为 900KB 的山羊图片，产生了一个影响性能的错误，以此作为一个测试用例。

这个示例 App 安装了（按字母排序）Crashlytics、Crittercism、Google Analytics 和 New Relic RUM 等工具。我使用的是这些软件的免费版本或可免费试用的版本。它们都报告了类似的信息，但是每一份报告仍然有略微的不同。所以，对于你的 App 来说，其中一个服务可能比其他服务更加适合。为了理解这些报告的数据，我们将查看这些工具的截图以便更好地了解可用于优化 App 的数据。

正如在第 2 章中讨论的，在测试耗电量时，有一些 RUM 数据会受到海森堡不确定性原理的影响。所有的 SDK 都会回传足够多的信息（你想要统计的）到服务器上。这可能会导致轻微的高数据流量和耗电。但这些 SDK 都对数据流量和耗电量做了良好的优化，我们将在 8.4.1 节中具体讨论 RUM SDK 的性能问题。

8.3 崩溃

正如在第 2 章所讨论的，性能是衡量用户是否满意 App 的关键因素。分析真实用户的设备

性能，能为你提供快速诊断和解决问题的能力。当提到性能时，我们通常从最关键的问题开始：App 的稳定性。通过崩溃日志，我们可以快速诊断并修复代码中的问题。

当加载过多图片时，将会发生下面的情况：

```
        imageViews=new ImageView[100];

    public int Imagelooper
     (int numberofaddedimages, int totalImageCount, RelativeLayout rl){
        for(int i=0;i<numberofaddedimages;i++)
        {

            totalImageCount = totalImageCount++;
            //为了分析而需要追踪崩溃……所以强制崩溃吧
           // if(totalImageCount ==100){
           //     totalImageCount=0;
           // }
                   //如果totalImageCount达到100, App会崩溃
                   //因为超出了数组边界

           imageViews[totalImageCount]=new ImageView(this);
```

当索引达到 100 时，超出了 `ImageView` 数组边界。Logcat 显示：

```
03-13 14:01:32.351 13772-13837/com.sillars.imagescroll I/
  image downloaded: number: 99
03-13 14:01:32.469 13772-13837/com.sillars.imagescroll I/ImageDownloader:
                              image99responsetime (2RTT): 38
```

下载第 99 张图片来回花费了大约 34 毫秒。这时，可能需要增加数组的边界：

```
03-13 14:01:34.637 13772-13772/com.sillars.imagescroll E/AndroidRuntime:
        FATAL EXCEPTION: main
    Process: com.sillars.imagescroll, PID: 13772
    java.lang.ArrayIndexOutOfBoundsException: length=100; index=100
            at com.sillars.imagescroll.MyActivity.Imagelooper
              (MyActivity.java:327)
            at com.sillars.imagescroll.MyActivity$3.onScrollStopped
              (MyActivity.java:178)
            at com.sillars.imagescroll.MyScrollView$1.run(MyScrollView.java:37)
            at android.os.Handler.handleCallback(Handler.java:739)
            at android.os.Handler.dispatchMessage(Handler.java:95)
            at android.os.Looper.loop(Looper.java:135)
            at android.app.ActivityThread.main(ActivityThread.java:5221)
            at java.lang.reflect.Method.invoke(Native Method)
            at java.lang.reflect.Method.invoke(Method.java:372)
            at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
                      (ZygoteInit.java:899)
            at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:694)
```

事实上，我们却得到一个数组的越界异常（长度是 100，索引值为 100）：

```
03-13 14:01:35.329  13772-13797/com.sillars.imagescroll I/Fabric:
                Crashlytics report upload complete: 35CC-27DD9B9DA026.cls
```

```
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: connected
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 102 HTTP transactions.
03-13 14:01:54.291  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 1 HTTP errors.
03-13 14:01:54.292  13772-15861/com.sillars.imagescroll I/
   com.newrelic.agent.android: Harvester: Sending 0 activity traces.
03-13 14:02:05.070  13772-13772/com.sillars.imagescroll I/Process:
                         Sending signal. PID: 13772 SIG: 9
```

在 App 发生异常后，报告中就多出了几条崩溃日志——崩溃信息被立即上传到了 Crashlytics 和 New Relic。（通过网络监控）可以看到，Crittercism 的报告通常发生在 App 退出后的一小段时间内。

在可控的测试环境下，手机或分析设备可以重现错误是一件很棒的事情。因为并不总是有这样的条件，所以让我们看一下这些工具都报告了什么。所有的 App 都以类似的方式报告崩溃，我们来看看 Crashlytics 的示例报告。

8.3.1　分析Crashlytics的崩溃报告

当 Crashlytics 发现新的崩溃时，你会立刻收到一个关于新问题的电子邮件报告（提示：建立一个接收崩溃报告的特定电子邮件或添加过滤器）。点击邮件中的链接可以跳到另一个网络页面上查看崩溃的详细信息。图 8-3 显示了 Imagelooper 崩溃的一个截图。

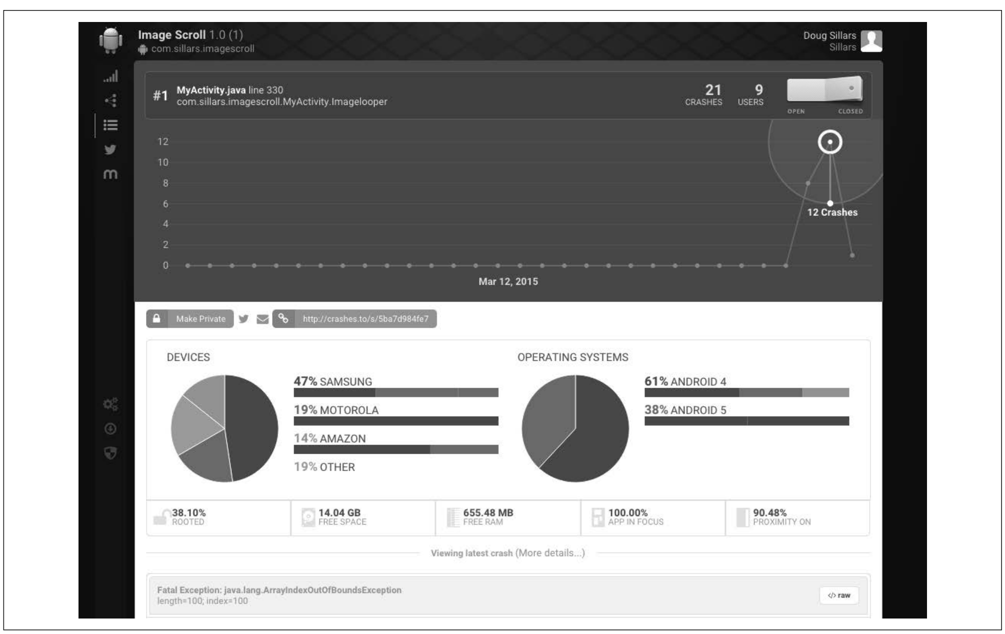


图 8-3：崩溃详情

在信息显示板的顶部显示了崩溃数量和用户数（在图中，9 个用户发生了 21 次崩溃），图中显示了最近三天的崩溃量（2015 年 3 月 12 日发生的 12 次崩溃被高亮显示）。页面中间的饼图和条形图根据 OEM（左）和 Android 版本（右）进行了区分。点击任意图表将会显示更详细的划分（在本例中，可以看到 Samsung 设备和运行 Android 4 的设备的数量），你可以更进一步地细分设备（OS 版本同理），见图 8-4。

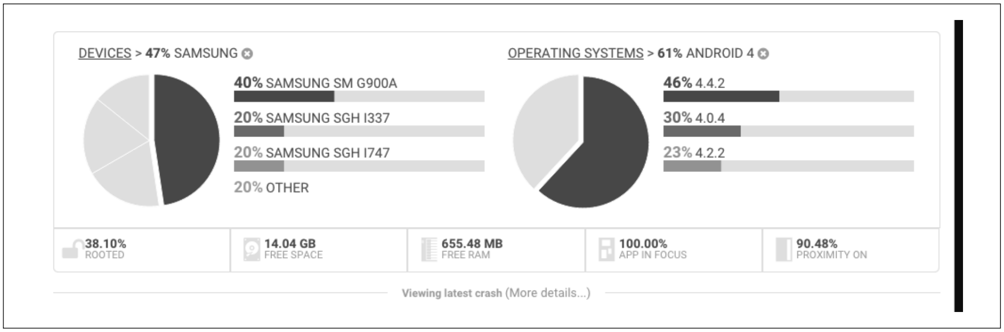


图 8-4：设备崩溃分析：Samsung 设备和 Android 4 的 OS 版本

饼图下面显示了崩溃发生时大概的设备详情：设备是否被 root 了？有没有可用的磁盘空间和内存？ App 是否正在前台运行？是否正在定位？下面是这个异常日志中的的栈信息。你可以查看最近的崩溃，通过接口获取精确的设备细节（包括所有的活动线程）。我已经将这个崩溃公开分享出来，有兴趣的话，可以查看这些细节：http://crashes.to/s/5ba7d984fe7。

远程跟踪崩溃使调试变得更加容易。这些工具要么提供跟踪 bug 的接口，要么提供从 bug 跟踪库中导出错误的方法。然而被动的响应崩溃并不理想，用这些工具可以查看影响用户最多的崩溃并将解决方案按优先排序。

如果你已经解决了前一节中崩溃报告列出的问题，那么现在就可以调查 App 在世界各地的运行情况了。这些工具可以报告 App 在全世界不同设备和网络上的表现，还可以隔离问题，并找到传统测试不能发现的 App 的瓶颈。也许你的 App 在一款中东流行的设备上发生了崩溃；或者突然出现了很多网络缓慢的非洲用户；又或者这些工具可能会收集到一些意料之外的信息，而这些信息在 App 未来的版本中可以被利用。许多的 SDK 和工具都会报告上述信息。图 8-5 显示了过去 24 小时内 Crittercism 报告的 App 使用情况。

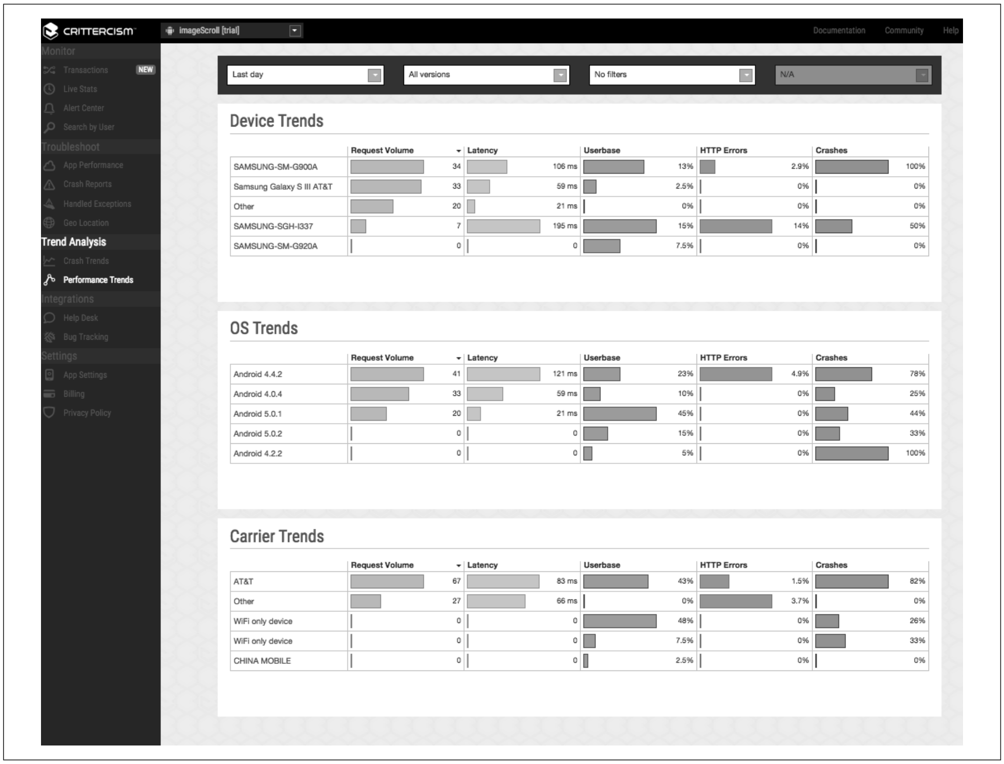


图 8-5：Crittercism 报告：24 小时使用趋向

这个信息显示板分别以流行的设备（顶部）、OS 版本（中间）和运营商（底部）的形式分析了请求数量、延迟毫秒数、用户群百分比、HTTPS 错误和崩溃百分比。从这个表来看，似乎大多数的 Samsung 设备都存在不同程度的延迟、错误和崩溃问题。这使你可以快速地查看 App 在特定的设备或 OS 版本上是否存在问题。如果一组设备（或者 OS 版本）表现出较长的延时或崩溃，你可以针对这些用户进行调查并推送修复补丁。

无线运营商的图表可以粗略地显示用户的分布位置。在这些工具中，也可以按照国家显示 App 在全球的用户分布。但由于这些数据是我和另外一位在 Twitter 上认识的上海人（非常乐于助人的人）一起生成的，这些地图非常枯燥。

来自上海的连接非常有趣，它显示这个连接非常缓慢，并且在传输过程中存在大量的错误。下面是来自 New Relic 的信息显示板（连接发生在太平洋时间 3 月 12 日的午夜）。在页面的主图上，我们发现一次连接花费近 20 秒（并且其他所有连接几乎不显示在页面上）。图下方的颜色表示，是网络异常导致了该问题。通过观察 App 随时间推移的运行情况，你会发现，网络或服务器延迟会出现在高峰时期，这表明服务器已经过载或该文件使网络负载过重。

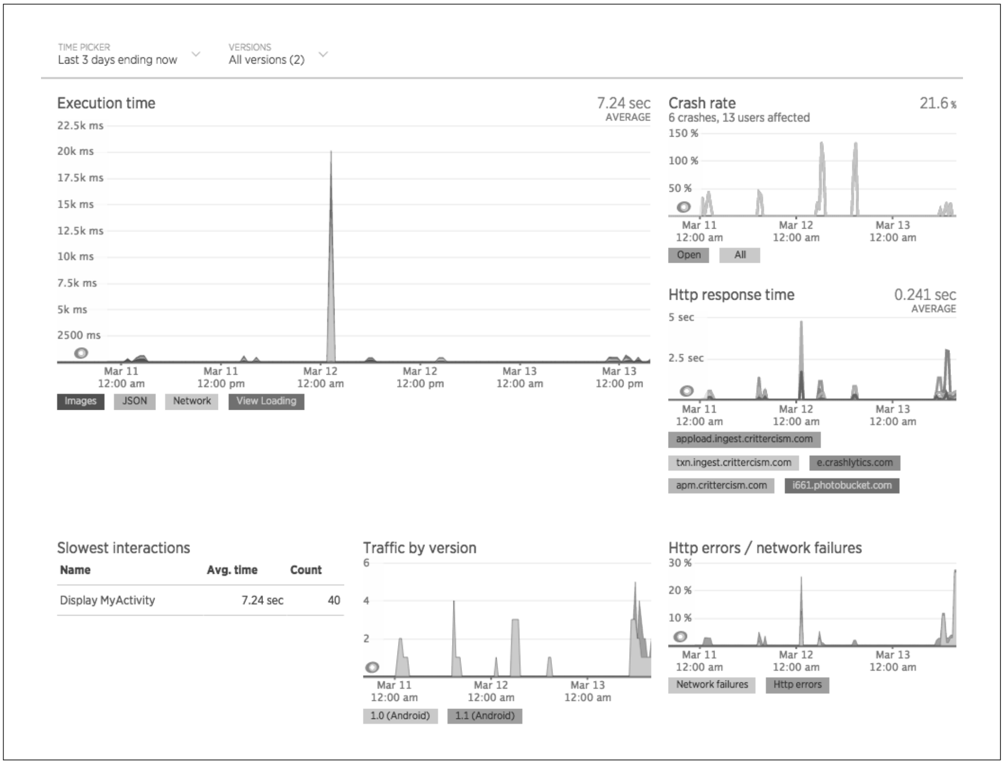


图 8-6：缓慢连接的信息显示板（另见彩插）

这个来自中国的连接的执行时间远远长于其他地方的连接。这可能仅仅是一个随机的异常。但如果之后继续看到某区域的缓慢连接，那么就有进一步调查的需要了。在显示板右侧中间的图是 HTTP 响应时间。为了连接到中国，Crittercism 响应时间几乎都接近 5 秒（没有面向客户，因此没有关系）。但是 Photobucket 的响应时间是 1720 毫秒。该显示板还包括随时间变化的崩溃率图（图中和 bug 状态匹配的颜色），以及随 App 版本和 HTTP 错误变化的流量。

有些工具可以通过域名查看延迟，如图 8-7 所示（服务器是否运行正常）。运营商的过滤器还可以查看某些地区获取数据的速度是否够快（在 7.4.2 节中提到过，Facebook 发现了某些国家特有的问题）。如果你注意到这些特点，就可以进一步确定 App 运行缓慢的地方。在下面的图表中，延迟峰值平均为 200 毫秒，但更深层次的调查显示，当图片提供者（Photobucket）的响应时间为 23 毫秒时，更慢的连接是解析。错误率出现的主导因素是，加载这 100 张图片中的 2 张拥有错误 URL 图片时抛出了 404 错误。这里看到的任何错误都值得我们更深入地剖析。

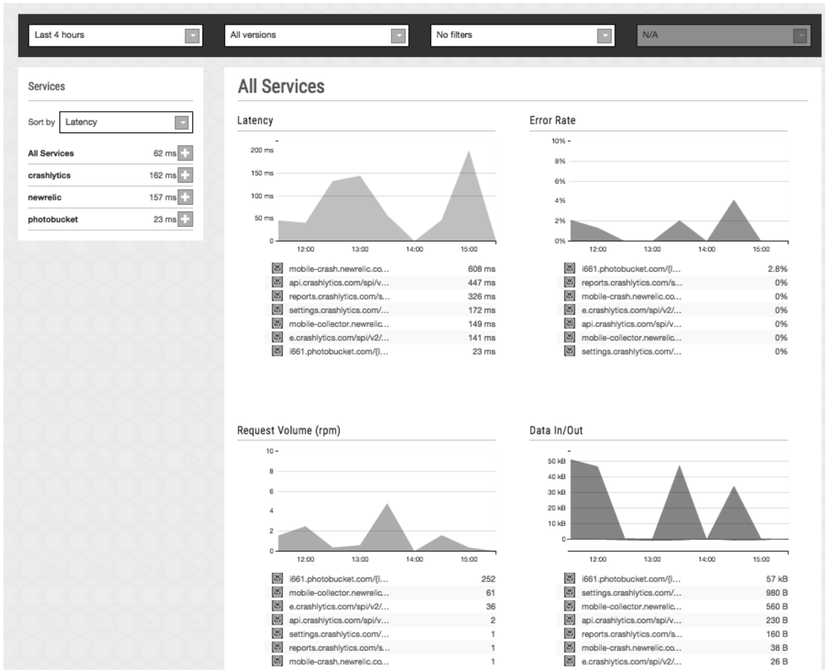


图 8-7：延迟、错误、声音和数据的图表

New Relic 工具有一个展示所有网络错误的列表。在图 8-8 中，可以看到所有来自 Photobucket（图片的 host）的问题都是 404 错误。点击即可查看来自 Photobucket 域名的 404 错误列表，加以确认，然后就可以在后台解决这些问题了。

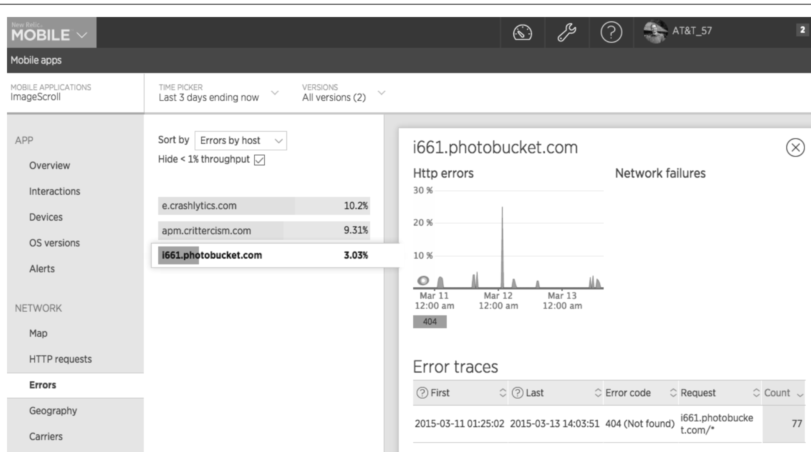


图 8-8：网络错误

通过服务器的日志关联 App 的 HTTP 错误，可以让你拥有非常强大的故障排除能力。你可以将问题范围缩小到可能引起麻烦的某个平台或版本。

8.3.2 使用

除了崩溃和性能，RUM 数据同样展示了很多关于用户的其他信息：用户是如何使用 App 的（以及使用的时长）、使用的频率，以及更多信息（见图 8-9）。通过更好地理解用户群，以及他们使用 App 的方式，你将拥有更多提升 App 的机会。

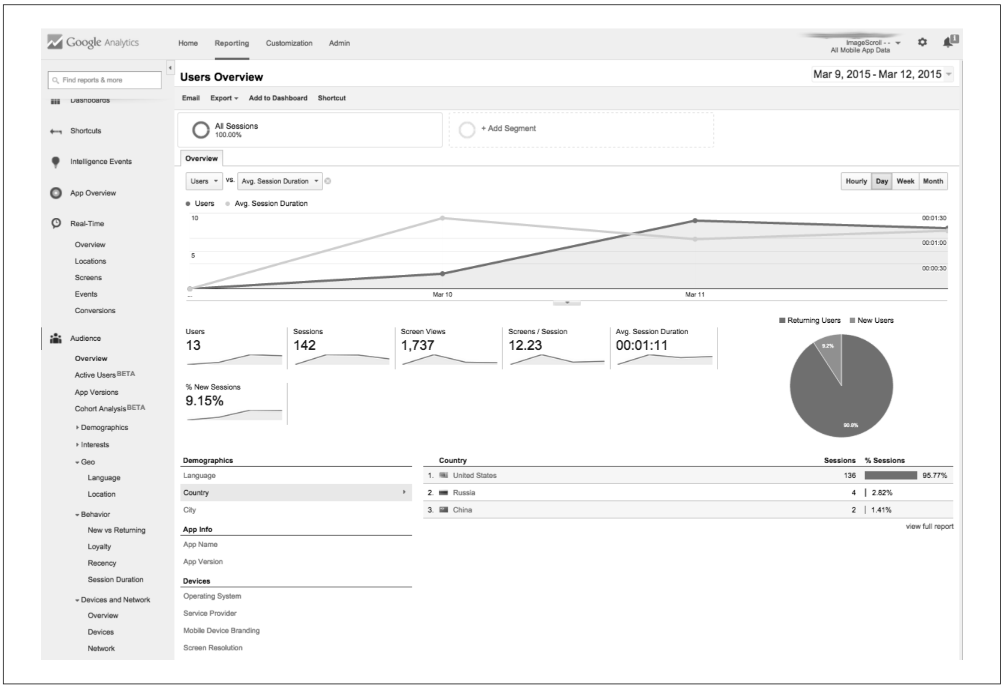


图 8-9：使用信息

Google Analytics 数据为我们提供了用户的相关信息。顶部图表显示了每日用户数量与平均会话时长的对比（3 月 11 日，拥有 9 个用户，平均会话时长为 59 秒）。其中共有 13 个用户，142 次会话（通过饼状图发现 91%的会话来自回头客）。Google Analytics 可以向用户显示特定的屏幕。通过这些数据可以收集相关信息，例如，什么页面导致用户退出 App 的，你可以在什么地方提升新用户流量。作为只有一个页面的 App，我将每 10 张图片标记为一个页面：

```
//在屏幕顶部启动追踪器
t = analytics.newTracker(R.xml.app_tracker);
        //启用广告功能
        t.enableAdvertisingIdCollection(true);
        t.enableExceptionReporting(true);
```

```
        t.setScreenName("top of scroll");
        t.send(new HitBuilders.ScreenViewBuilder().build());
<snip>
//另外10张图片被请求,所以在Google Analytics中更新屏幕名称
                    t.setScreenName(totalImageCount + " images");
                    t.send(new HitBuilders.ScreenViewBuilder()
                            .build());
//加入一个crittercism Breadcrumb
                 Crittercism.leaveBreadcrumb(totalImageCount + " images");
```

在 App 中加入这些代码段后，每当用户滚动 10 张图片时，Google 将会添加一个页面视图，Crittercism 将会添加一个 breadcrumb 以跟踪问题。让我们看一下来自 Google Analytics 的显示报告（见图 8-10）。表中显示了你可能期望看到的内容，App 启动页可能是最常见的（“Doug Scroll App”是页面的启动页名称），然后当用户滑动初始页进入 App 页面时，又在不同的时间里退出。大部分退出的用户并没有任何滚动操作，也有很多用户在滚动到 90 张图片时退出（归咎于 99 张图片后导致 App 崩溃的 bug）。另外一个有趣的特征是，在一个视图中停留平均时间最长的是第 90 张图片。App 发生了 ANR 而不是崩溃，这使 App 停滞在当前页，然后加长了这个页面的使用时间。

Primary Dimension: Screen Name

Plot Rows | Secondary dimension | Sort Type: Default | advanced

Screen Name	Screen Views	Unique Screen Views	Avg. Time on Screen	% Exit
	1,737 % of Total: 100.00% (1,737)	702 % of Total: 100.00% (702)	00:00:06 Avg for View: 00:00:06 (0.00%)	8.12% Avg for View: 8.12% (0.00%)
1. 10 images	283 (16.29%)	15 (2.14%)	00:00:01	0.00%
2. 20 images	283 (16.29%)	77 (10.97%)	00:00:03	2.12%
3. Doug Scroll App	223 (12.84%)	137 (19.52%)	00:00:13	17.04%
4. 30 images	180 (10.36%)	68 (9.69%)	00:00:07	0.00%
5. 40 images	142 (8.18%)	60 (8.55%)	00:00:08	1.41%
6. top of scroll	95 (5.47%)	79 (11.25%)	00:00:00	65.26%
7. 50 images	85 (4.89%)	56 (7.98%)	00:00:07	4.71%
8. 60 images	79 (4.55%)	52 (7.41%)	00:00:04	3.80%
9. 70 images	64 (3.68%)	49 (6.98%)	00:00:11	6.25%
10. 80 images	63 (3.63%)	42 (5.98%)	00:00:03	1.59%
11. 90 images	53 (3.05%)	40 (5.70%)	00:00:34	24.53%

图 8-10：App 中的视图

对于真实的 App 来说，在特定页面上花费的时长可以表明用户是如何与 App 进行交互的。除了可以统计出每个页面所用的时间，Google Analytics 还可以分析出页面间的流动性。在示例 App 中，很明显可以看出大部分用户会从 10 张图滑到 20 张图等。对于一个复杂的 App 来说，通过数据流和视图可以帮助你发现用户发现不了的问题。如果观察设备或页面大小，发现缺少了页面，也许是点击之后系统渲染的方式出现了问题，从而导致用户无法按照预期的方式使用 App。有多种方法可以用来研究数据流，其中一种是将所有用户分为

较小的子集。在图 8-11 中，所有数据流通都被标记为灰色，而更深的线则表示来自美国加州用户的数据流通。

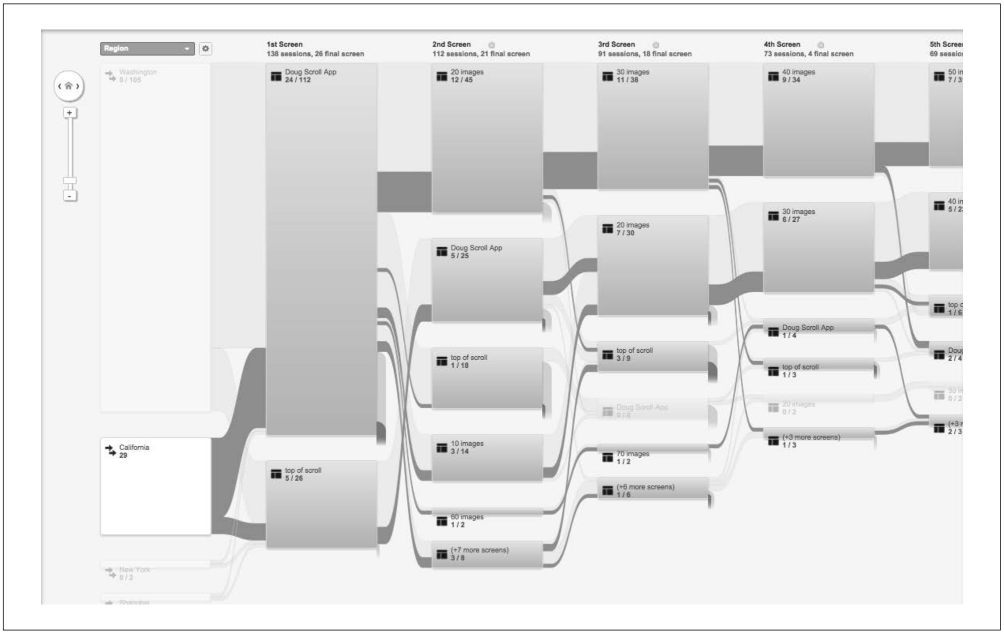


图 8-11：行为流（加州用户是被高亮的）

我们还可以将设置的唯一事件上报到服务器以发现问题。在第 7 章中，我们用例 7-4 识别用户使用的网络类型，用例 7-5 修改可用网络带宽传过来的内容。我们把这种逻辑使用到 Image Scroll App 中，并将分析引擎发现的网络类型和 RTT 时间上报。

```
t.send(new HitBuilders.EventBuilder()
              .setCategory("RTT Event")
              .setValue(AvgRTT.longValue())
              .setAction("ImageRTT").setLabel(networkConnection).build());
Crittercism.beginTransaction(networkConnection);
Crittercism.setTransactionValue(networkConnection, AvgRTT.intValue());
Crittercism.endTransaction(networkConnection);
```

这个报告数据如图 8-12 所示。

Primary Dimension: Event Category Event Action **Event Label**

Plot Rows | Secondary dimension | Sort Type: Default | advanced

	Event Label	Total Events	Unique Events	Event Value	Avg. Value
		420 % of Total: 100.00% (420)	67 % of Total: 100.00% (67)	19,981 % of Total: 100.00% (19,981)	47.57 Avg for View: 47.57 (0.00%)
1.	Wi-Fi	308 (73.33%)	50 (71.43%)	11,361 (56.86%)	36.89
2.	HSPA+	83 (19.76%)	15 (21.43%)	7,472 (37.40%)	90.02
3.	RTT Event	22 (5.24%)	1 (1.43%)	390 (1.95%)	17.73
4.	HSPA	6 (1.43%)	3 (4.29%)	678 (3.39%)	113.00
5.	LTE	1 (0.24%)	1 (1.43%)	80 (0.40%)	80.00

图 8-12：以 App 区分的网络类型

每当页面刷新时，检查网络类型，这样就收集了数以千计的 RTT 时间（第三列显示获得了将近 20 000 个值）。平均往返时间可以在图 8-12 的最后一行中看到。平均往返时间可能并不是非常有用，因为它会随着信号的强度、位置以及网络的类型变化。然而，运用第二维度测量数据时，可以根据设备、网络类型、地铁区域和陆地获取数据，用多种方法对信息进行切割和筛选研究。图 8-13 根据美国的地铁区域进行排序，结果显示，西雅图 Wi-Fi 的 RTT 比洛杉矶和纽约更加快速。

Primary Dimension: Event Action **Event Label**

Plot Rows | Secondary dimension: Metro | Sort Type: Default

	Event Label	Metro	Total Events	Unique Events	Event Value	Avg. Value
			420 100.00% (420)	67 100.00% (67)	19,981 100.00% (19,981)	47.57 47.57 (0.00%)
1.	Wi-Fi	New York NY	16 (3.81%)	2 (2.86%)	2,589 (12.96%)	161.81
2.	Wi-Fi	San Francisco-Oakland-San Jose CA	22 (5.24%)	4 (5.71%)	1,595 (7.98%)	72.50
3.	Wi-Fi	Seattle-Tacoma WA	270 (64.29%)	44 (62.86%)	7,177 (35.92%)	26.58
4.	RTT Event	Seattle-Tacoma WA	22 (5.24%)	1 (1.43%)	390 (1.95%)	17.73
5.	LTE	San Francisco-Oakland-San Jose CA	1 (0.24%)	1 (1.43%)	80 (0.40%)	80.00
6.	HSPA+	San Francisco-Oakland-San Jose CA	67 (15.95%)	13 (18.57%)	6,747 (33.77%)	100.70
7.	HSPA+	Seattle-Tacoma WA	16 (3.81%)	2 (2.86%)	725 (3.63%)	45.31
8.	HSPA	San Francisco-Oakland-San Jose CA	6 (1.43%)	3 (4.29%)	678 (3.39%)	113.00

图 8-13：以城市区分的网络类型

命名页面，增加自定义事件和计时器，可以创建用户使用 App 的详细图，从而找出加载较慢的页面、丢失的导航数据，以及其他的用户流问题；可以发现世界上是否有某些特定的区域会出现更多的延迟、错误或其他比较慢的表现；还可以发现在特定的日期或一天的某

个时段内，由于网络阻塞（甚至自己的服务器阻塞）导致的性能问题。

8.3.3　实时信息

因为这些上报的 SDK 定时地收集分析数据，所以用户追踪可以说是实时进行的。以上讨论过的所有提供商基本上都能实时显示 App 的性能。在图 8-14 中，可以看到在最近的 30 分钟内，启动了 4 个 App，出现了一次崩溃。

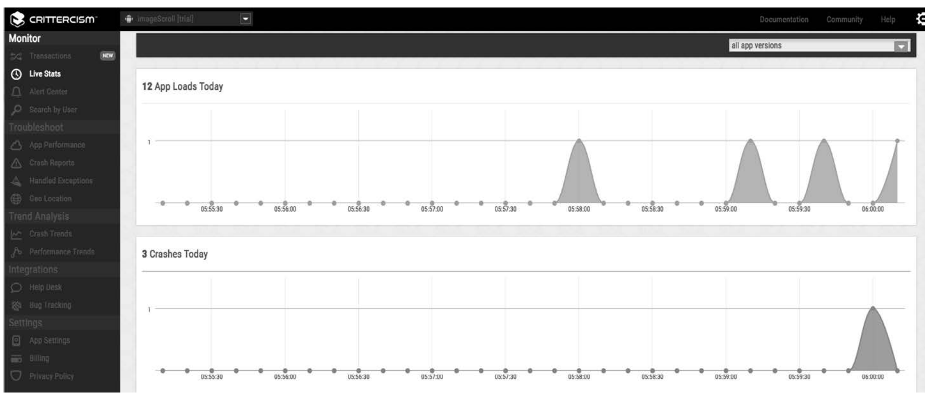


图 8-14：实时分析

8.4　大数据的营救

当 App 不断地迭代并且获得越来越多的用户，迅速确定并解决问题变得更为重要，也更加困难。找出哪些问题影响了大多数的用户以及问题的严重性，可以在解决 bug 时更有目的性。本章所涉及的工具有助于压缩数据，发现有用的模式和需要优化的问题，这样可以精简 App 并使用户使用时更加顺心。

好的 RUM 可以确保升级版本比以往版本在性能上表现得更好、崩溃量更少，加载速度更快。利用 RUM 工具进行大数据的收集仍然是一种响应式的解决问题的方式，制订周全的计划可以提升对 App 性能的洞察力，并了解性能提升是如何提高 App 保留率和用户使用时间的。

RUM SDK的性能

尽管这些 SDK 的初衷是为了测量性能，但测量这些工具的性能也是一个不错的想法。如果你注意到先前的截图，每一个 SDK 都报告了其他 SDK 的延迟（而不是自己的）。对于用户数据来说，保持较短的往返时间非常重要，而对于之后的文件存取，时间稍长也是可以的。如果看到大量的 HTTP 连接错误，你可能会开始担心。

为了测试监控 SDK 的网络性能，可以使用 7.2.4 节中提到的应用资源优化器——AT&T ARO。ARO 可以根据端点筛选数据。

图 8-15 在顶部显示了完整的 App 数据，在底部显示了数据分析的视图。

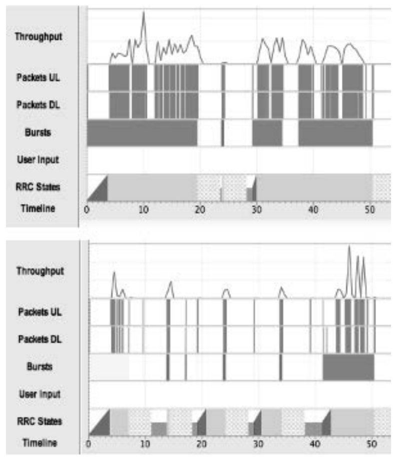


图 8-15：Image Scroll 的网络追踪：完整 App（顶部）和仅 RUM 连接（底部）

来自供应商的 RUM 文件都是加密的，为了看到它们，我做了同 7.2.3 节中相同的测试。分析供应商的 RUM 数据并不危及任何用户的数据，而且这些数据是预料中的。下面是 Crittercism 的一个崩溃报告：

```
2015-03-16 13:38:41 POST https://api.crittercism.com/android_v2/handle_crashes
                        ←200 application/json 14B 46.33kB/s
Request
Response
x-newrelic-id:    <removed>
Accept:           text/plain
Accept:           application/json
Content-Type:     application/json
User-Agent:       5.0.6
Host:             api.crittercism.com
Connection:       Keep-Alive
Accept-Encoding:  gzip
Content-Length:   28095
JSON
{
    "app_id": "<removed>",
    "crashes": [
        {
            "app_state": {
                "activity": "com.sillars.imagescroll.MyActivity",
                "app_version": "1.1",
                "app_version_code": 2,
```

```
"arch": "armv7l",
"battery_level": 0.16,
"carrier": "",
"disk_space_free": "11018395648",
"disk_space_total": "24723058688",
"dpi": 3.5,
"locale": "en",
"memory_total": 268435456,
"memory_usage": 90950656,
"mobile_country_code": 0,
"mobile_network": {
    "available": true,
    "connected": false,
    "connecting": false,
    "failover": false,
    "roaming": false
},
"mobile_network_code": 0,
"model": "Nexus 6",
"name": "",
"orientation": 1,
"sd_space_free": "11018395648",
"sd_space_total": "24723058688",
"system": "android",
"system_version": "5.0.1",
"wifi": {
    "available": true,
    "connected": true,
    "failover": false
```

我们可以看到 App、版本、低电量（16%）、有很多的空闲磁盘空间和内存、正在使用 Wi-Fi（但是移动网络可用）等信息。信息板上的所有这些数据对诊断崩溃都是有帮助的，在收集的所有文件中，没有传输预想不到的数据。

8.5 小结

在本章中，我们研究了如何从用户端收集 RUM 分析数据，这有助于确定不能测试到的设备问题。通过日志追踪设备的崩溃，可以直接解决问题而不用每次都拿着设备分析问题。

这些数据同样可以帮助你揭露地域性问题，比如在世界特定区域内网络连接缓慢的问题。追踪数据，你可能会发现用户用意料之外的方式使用 App，而为了满足这些新的用法，精简用户流会使 App 体验更好。

通过使用分析工具武装 App，你可以获取更加有力的数据以分析 App 用户量下降的原因，需要提升的地方，以及用户监控下什么部分运行得很好。用真实设备从真实用户那里获得的真实数据是无价的，它们可以用于提升 App 性能——修复所有你在实验室不能发现的问题点。

附录

组织性能

为了成功地优化 Android App 的各个方面的性能，本书能够帮助你让整个公司重视性能的问题。开发者、测试人员以及公司管理人员需要达成共识：高性能对于 App（甚至公司的成功）来说是至关重要的。一旦形成团队，就需要制定流程以确保正在做的开发和 App 维持了稳定的性能指标。最后，我们将回顾一些本书中略述的工具以作为性能优化流程实现的一部分。

开始支持（管理人员关注性能）

要让 App 的性能成为公司文化的一部分，获得管理人员的支持是至关重要的。这里有很多关于慢速网站性能的数据，以及少量同样有变慢趋势的移动 App 的性能数据。因此，如果你无法说服公司的管理者，让他认识到 App 性能对公司命脉是多么至关重要，那么，你可以参见本书 1.1 节和 1.1.3 节，那里的数据也许可以让你的领导改变观点——什么公司不想降低成本，增加收入呢？或许《纽约时报》上的低性能是如何成为击沉 Friendster（一个社交媒体先驱）的一个因素的研究案例能够帮助到你。

将这些信息同潜在的问题联系在一起，并提出优化方案通常是开启性能谈话的一个很好的方式。由于修复程序已经完成，而且收益从使用量、用户参与度和销售额中是可见的，那么扩大正在运行中的性能优化团队是一件很容易的事。如果你是组织中的第一个人，那么你将可能开始成为测试和发现问题的人。

Steve Souders 发布的“创建性能文化”（http://calendar.perfplanet.com/2012/creating-a-performance-culture/）的帖子中有一个重要的观点——说适合听众的言词。如果你正在试图

说服市场人员，那么你应该说增加用户、参与度和销售额方面的东西，运营人员则希望听到容量或者减少故障方面的变化，而财务人员则很想听到成本降低的同时销售额的增长。通过稍微改变听众听到的关键字，我们会发现获得支持变得更加容易了。

在 AT&T 公司，与领导讨论性能时，我们是非常幸运的。他们认识到，拥有高性能的移动 App 能够减少数据使用量，拥有更长的电池寿命，并且最终会让用户更加满意。其结果是，AT&T 公司已经为所有内部开发的 App 以及所有预装到我们设备的 App 制定了性能测试要求，并且我们将会持续向公司内外的开发者推广这些测试。

浅谈性能

2011 年年初（想想中期姜饼时代），我们开始了 AT & T 的应用程序资源优化工作。我们开始关注 Android App 是如何使用数据的，并且惊讶于它们竟是如此的低效。当和开发者讨论时，我们才意识到并没有人真正地考虑过移动数据的性能。我们发现，这个例子中确实存在 80/20 规则。80% 的时间里，如果开发者关注到了 App 的行为，他们将努力地优化此 App。剩下的 20%可能是被故障卡住了，也许需要更多组织或者测试的帮助。

每当向其他团队提出性能问题时，我们应该尽量委婉一点。没有人会感激你提出这些问题，尤其是那些他们的监控不关注的事情。如果能够坚持积极地致力于 App 的潜在加速或者改进，你将会在性能的道路上获得更多的追随者。

Lara Hogan 写到如何成为团队的的性能守护者 / 看门人（https://davidwalsh.name/performance-cops-janitors），以及性能是如何导致资源耗尽的。她认为，性能领导是必不可少的，但他们应该努力地在全公司实行流程规范，让性能成为日常工作中必不可少的一部分。作为性能推广团队的一部分，我们为合作的公司充当着守护者的角色。有些公司的开发者和管理者对性能有一定了解，并且也会测试性能，但是由于开发者承担了更多其他的责任，性能测试就要为之让路。但每隔几个月向他们发送有关性能的友好提醒，会让他们步入正轨。

在 2013 年的 Oredev 上，Scott Barber 说了一个故事，这个故事是关于一个没有优化和性能测试预算的项目的。他要求前台的管理员一周向开发者询问一次 App 的性能，他发现，一个简单的关于性能的提醒就能够让开发者在开发的过程中意识到性能问题，并最终缩减了 App 的加载时间。阅读关于性能的博客，并与你的同事分享小花絮。当发现新的性能技术时，在组织内外分享它。通过帮助别人学习如何使移动 App 变得更快，你将能激发并激励团队，以及那些和你一起工作的人。

通过使性能成为组织定期谈话的一部分，你会开始定期发现性能有所改进和提升。你应该经常谈论性能，分享其他团队（公司外部）已经分享过的成功案例，也可以分享组织内部的大提升。我们也会遭遇挫折。App 可能会在启动时出现问题。诀窍是尽可能快地定位问题，并解决问题。下面的几节内容会涵盖我们发现的一些有效策略。

发展

我们都知道这个格言 / 笑话：最好的代码是没有代码。只要在屏幕上敲出第一个代码，App 就会变得比前一刻更慢。当代码被改变、改善或者添加时，我们应该考虑这对用户性能所产生的影响，要将开发最小化，并进行最终测试。

如果开发人员考虑到性能的话，那么他们将会努力地确保每一个新功能都是以一种无缝的方式添加的。这并不总是发生，哪怕是在一个产品中测试 App 的性能，AT&T 的应用资源优化小组也已经发现了在构建 App 时不考虑性能的案例。

曾经，我们在 ARO 工具中添加了一个最佳实践，开发人员并没有刻意地将这个功能同已有的代码整合在一起，仅仅只是添加了代码，结果就导致 App 多次扫描了多兆字节的网络记录，分析时间显著增加了。

作为快速发展的结果（性能工具是必不可少的！），我们开始研究 App 的每一个变化对性能的影响。我希望能够说这一环节是全自动的，或者我们是用秒表的时间来衡量 App 的变化对性能的影响，但事实并非如此。每当增加代码时，我都会对代码进行比较，以确保性能成本是可接受的。在我们的团队中，需求提出者同开发人员密切合作，并且能够经常看到粗糙版本的工具和功能，允许我们就 UI、布局、语法以及（当然）性能提出自己的观点。

在 AT&T 内部，ARO 推广小组扮演着性能问题支持团队的角色，我们收到了 AT&T 内部不同组织的，关于内外部 App 的帮助请求。通过帮助这些开发者发现常见的问题，他们往往能够快速地解决那些拖慢 Android App 的问题。拥有理解并有权研究和解决性能问题的开发团队，可能是一项挑战（涵盖所有的新特性、bug 和技术积累，这是一项非常艰巨的挑战），但是解决关键的性能问题将会真正提高公司的短板。

测试

是的，测试永远是截止日期快到的时候以及开发时间被压缩的时候，第一个被削减的环节。如果没有性能测试，你通过分析可能只会发现新功能使得 App 速度减慢。但因为你已经在产品中加入了这个会减慢 App 的功能，用户会认为你整个 App 都是低效的，正如我在 1.2.1 节中所描述的情形一样。

当添加了新的功能，毫无疑问的是必须要进行测试，以确保该功能可以正常工作，并确保其不会破坏 App 的其他部分。如果只是测试崩溃，其实就是在处理*正在流失的*性能，要确保新的代码不会引入延迟或减慢速度。它们可能会引出 App 非常多的崩溃。如果新功能导致超出预期的缓慢，那么需要确定在速度方面的变化是否是可以接受的，或者确定是否可以将该功能撤回以进行进一步优化。

测试提示

AT & T，应用程序质量联盟（Application Quality Alliance，AQuA）的合作伙伴，给出了一些测试网络性能的最佳实践。测试用例是开始整套性能测试的一个很好的起点。如果你有很棒的性能测试用例，可以将它们分享给我们，我们将与其他开发者共享。

性能指标

当涉及性能，应该由团队共同决定正确的速度指标。现在有许多关于用户对网络期待的研究（聚焦 App 的研究也在不断增多）。做自己的测试，然后查看用户所期望的是什么，以及他们是否发现你的 App 速度缓慢。如果发现 App 很慢，你需要确定设备实验室（第 2 章）中的哪些参考设备比其他设备慢，并在开发过程中使用它们进行测试。

移动 App 是如此多样和独特，以至于构建一个适用于所有 App 的测试用例几乎是不可能的。测试用例对流媒体 App 来说是必不可少的，但是不适用于社交 App、游戏 App 或新闻 App。我分享的测试用例是非常通用的，可以对特定的 App 进行测试。对于你的 App，你可以按照常识和团队商讨出测试用例，然后将它们制定为规则使大家坚持下去。当某个指标超出了规范，确定是不是有 bug 产生，然后尽可能快地（理想情况下是在发布前）解决这个问题。

测试性能指标

用户最经常抱怨的就是 Android 设备的电池寿命。在过去，这些罪名通常落到了设备制造商或电池的缺陷上，但现在用户逐渐明白了 App 也会引发此类问题。在第 3 章中，我们讨论了如何确定 App 电池耗尽的原因，从 WakeLock 到设备的无线电的过度使用。我们还分析了 Lollipop 版本的 Jobscheduler API 作为新设备的解决方案的可能性，以及如何使用 Battery Historian 来查明在 App 中引起电池消耗的问题。

用户和 App 在屏幕上进行交互，页面卡顿不流畅是用户放弃 App 的首要因素。通常情况下，对 App 速度的感知会影响 App 的使用量，在第 4 章我们研究了如何简化 UI 层级，以及如何使用 Systrace 和其他工具来测试 UI 的卡顿和速度问题。如果 App 由于内存泄漏或 ANR 问题引起了崩溃，使用 MAT 或 Traceview（参见第 7 章）等工具将帮助你找出问题产生的原因，让你可以回到代码中去解决问题。

移动开发中另一个会显著增加延迟的因素是网络连接。虽然你无法控制用户的位置，或者他们连接的网络，但是你可以优化 App 传输的数据量，以确保体验始终运行流畅。在第 6 章中，我们介绍了网络连通性，简化数据使用的技巧，以及 Wireshark、MITMproxy、Fidler 和 ARO 等工具，从而测试并尽可能优化连接。最后，我们在第 8 章中论述了如何使

用用户监控工具从用户那里得到测试结果。通过了解用户的痛点在哪，你可以回过头来确保障碍被移除，死机问题得以解决，同时，还应该确保当前（和将来）用户在你的 App 里有一个良好的体验。

通过本书提到的理论和工具，现在你已经拥有了提升 Android App 性能所需要的所有东西。通过深挖这些工具和技术，你可以找到加快渲染，减少延迟和电池消耗的方法，并最终提高 App 的性能。

关于作者

Doug Sillars 是 AT&T 开发者计划中的性能推广领导者。他帮助了成千上万的移动开发人员将性能的最佳实践应用到 App 上。他开发的工具和总结的最佳实践，帮助开发人员使 App 运行得更快，同时使用了更少的数据和电量。他和妻子生活在华盛顿州的一个小岛上，并在家教育三个孩子。

封面介绍

本书封面上的动物是一只**玄燕鸥**（brown noddy）。这是海鸥和燕鸥家庭的一部分，主要栖息于温暖的热带水域。这种鸟也是一种非常常见的燕鸥。

玄燕鸥有着独特的外观。羽毛都是棕色的（因此才得名 brown noddy），但前额是灰白色的，并且一直延伸到眼睛的顶部。尾巴呈楔形，颜色是比身体的其他部分更深的深棕色。翅膀的翼尖也呈这种深棕色。腿、脚、喙也呈深棕色。雄性和雌性的玄燕鸥看起来相似，只是雌性体型稍小一点。

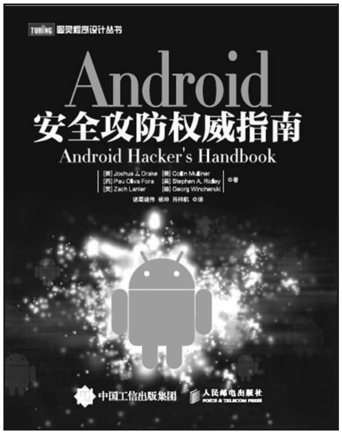
和大多数海鸟一样，玄燕鸥主要以所生活的热带水域和海域可以提供的食物为食。它们会在海面上捕食小鱼，收集漂浮上来的水下捕食者的猎物残渣，同时也会沿着海岸线和浅滩寻找食物。

玄燕鸥的筑巢和育种相当繁杂，并且主要是在近海。它们会在树、灌木、悬崖、海滩和码头筑巢。它们每年只产一个蛋，双亲会轮流孵化。这种亲子关系在鸟蛋孵化后仍会继续，直到雏鸟在八周大左右离开鸟巢为止。

O’Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

封面的图片来自 *British Birds*。

延 展 阅 读



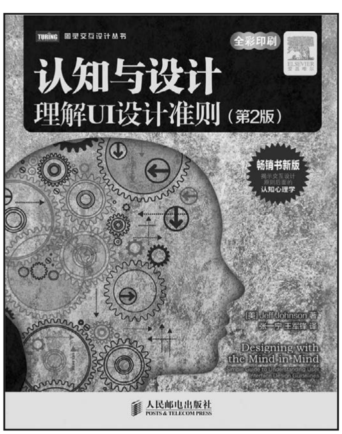
Android 安全第一书，专注于阐述设备 root、逆向工程、漏洞研究和软件漏洞利用等技术细节

书号：978-7-115-38570-3
定价：89.00 元



Amazon 榜首畅销书全新升级，Android 入门进阶不二之选
全面覆盖开发知识点，通过实战项目教你逐步写 Android 应用

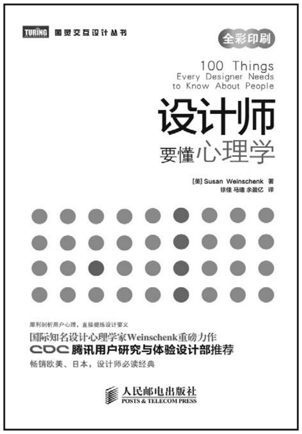
书号：978-7-115-42246-0
定价：109.00 元



将心理学基本原理与设计基本原则有机结合
揭示交互设计原则后面的认知心理学

书号：978-7-115-36410-4
定价：69.00 元

延 展 阅 读



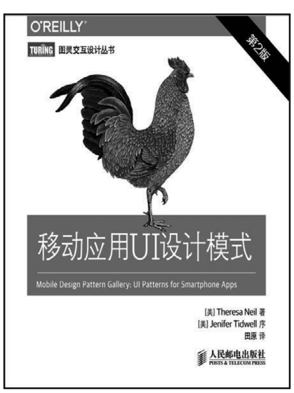
国际知名的设计心理学博士 Susan M. Weinschenk 重磅力作
腾讯用户研究与体验设计部（CDC）推荐
畅销欧美、日本，首屈一指的设计师必读经典

书号：978-7-115-31308-9
定价：49.00 元



超级畅销书《设计师要懂心理学》姊妹篇
国际知名设计心理学家 Weinschenk 全新力作
用讲故事的手法生动呈现 100 个设计案例
用户体验设计师 / 交互设计师 / 产品经理必读经典

书号：978-7-115-42784-7
定价：59.00 元



畅销手册全新升级，全彩印刷
赏心悦目、简单易用的权威 UI 设计参考书
90 多种设计模式，1000 张屏幕截图，一册在手，应有尽有

书号：978-7-115-37790-6
定价：79.00 元

O'REILLY®

高性能Android应用开发

对于Android App来说，虽然独特巧妙的构思至关重要，但速度、效率和电池管理才是App成功的真正驱动力。本书直面Android App开发中的性能问题，系统介绍App性能优化的方法和工具，旨在帮助Android开发人员正确构建快速、流畅的App，让其能够在近2万种各式Android设备上运行良好。

- **了解性能问题是如何影响App的用户量和留存率的**
- **构建Android设备实验室，进行最大限度的UI、功能和性能测试**
- **提升App和硬件设备的交互方式**
- **优化UI，使其能够快速进行渲染、滚动以及动画**
- **跟踪内存泄漏和对性能有影响的CPU问题**
- **升级App同服务器的通信，了解慢速网络条件下如何加强App体验**
- **应用RUM工具监控以确保每一个设备都拥有最佳的用户体验**

Doug Sillars，AT&T开发者计划中的性能推广领导者。他开发的工具和总结的最佳实践，已经帮助数万名移动开发人员将App运行得更快。

"这本书将使得任何Android开发者都能够构建高效、运行良好的App。"

——Brad Zeschuk
M2Catalyst公司工程副总裁

"本书是Android性能方面的权威实战指南，可以帮助工程师转换视角。书中不仅涵盖了基本的算法话题，还深入到了硬件和平台的工作方式，让你了解工具的异常显示是什么含义。"

——Colt McAnlis
资深布道师，
Google公司团队主管

MOBILE

封面设计：Karen Montgomery　马冬燕

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／移动开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-43570-5

9 787115 435705 >

ISBN 978-7-115-43570-5
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks